System/File No. 1620/1710-21
Form C26-5600-1

IBM Systems Reference Library

IBM 1620/1710 Symbolic Programming System

This manual describes details of the 1620/1710 SPS Two-Pass Processor, namely, statement writing, operations and associated mnemonic operation codes, preediting the source program, adding user's subroutines and macro-instructions, organization and operations of the processor, formats of both uncondensed and condensed card output decks, operating procedures, and special procedures (condensed object deck alteration, modifying the processor for additional storage, condenser program). A 7090 processor for assembling 1620/1710 programs is also described.

This manual is written for the following IBM Programming Systems:

1620-SP-020	1620/1710 SPS for card system
1620-SP-021	1620/1710 SPS for paper tape system
1710-SP-001	7090 Processor for Assembling 1620/1710 SPS Programs

This publication, a major revision of Form C26-5600-0, makes the prior edition obsolete. In addition to incorporating information released in Technical Newsletters N26-0007 and N26-0009, a significant change has been made to the section "Listing the Uncondensed Object Deck" by adding an IBM 407-E8 control panel wiring diagram.

POST-PUBLICATION CHANGE PAGE HISTORY

Technical Newsletter N26-0026 has been incorporated in this publication as directed by a new revision system. Some of the principal aspects of this new revision system are:

1. Changed or new text is identified by a vertical line in the margin on revision pages.
2. New or changed illustrations are identified by a large dot preceding the titles of figures or tables.
3. Each revision page number is dated for convenience of filing.
4. Added pages, figures, or tables are numbered n.1, n.2, n.3, etc.

The following pages were affected by the newsletter:

OLD PAGE	NEW PAGE
1 (cover)	1 (revised 4/1/63)
2 (this page)	2 (revised 4/1/63)
3 (contents page)	3 (revised 4/1/63)
5	5 (revised 4/1/63)
17	17 (revised 4/1/63)
53	53 (revised 4/1/63)
86	86 (revised 4/1/63)
88	88 (revised 4/1/63)
98	98 (revised 4/1/63)
101	101 (revised 4/1/63)
104	104 (revised 4/1/63)
112	112 (revised 4/1/63)
113	113 (revised 4/1/63)
114	114 (revised 4/1/63)
115	115 (revised 4/1/63)
116	116 (revised 4/1/63)
117	117 (revised 4/1/63)

Copies of this and other IBM publications can be obtained through IBM Branch Offices. Address comments concerning the content of this publication to: IBM, Product Publications Department, San Jose, California

Contents

Preface

The Symbolic Programming System is designed to simplify the preparation of programs for the IBM 1620 Data Processing System and the IBM 1710 Control System. The development of larger and more versatile data processing systems like the 1620 and 1710 has resulted in a greater number of, and more complex, machine language instructions. The difficulties of coding in machine language – a tedious and time-consuming task – have been recognized and one of the efforts at simplification is the system known as Symbolic Programming.

The Symbolic Programming System permits the programmer to code in a symbolic language that is more meaningful and easy to handle than numerical machine language. SPS automatically assigns and keeps a record of storage locations, and checks for coding errors. By relieving the programmer of these burdensome tasks, SPS significantly reduces the amount of programming time and effort required.

This manual is intended to serve as a reference text for the 1620/1710 Symbolic Programming System. It assumes the programmer is familiar with the methods of data handling and the functions of instructions in the 1620 Data Processing System. For those without such knowledge, information on 1620 and 1710 instructions may be found in the IBM *Reference Manual, 1620 Data Processing System* (Form A 26-4500) and the IBM *Reference Manual, 1710 Control System* (Form A26-5601).

IBM 1620/1710 Symbolic Programming System

Introduction

The 1620/1710 Symbolic Programming System may be divided into the symbolic language used in writing a program, the library containing the subroutines and linkage instructions (macro-instructions) that may be incorporated into the program, and the processor program that is used to assemble the user's program.

Symbolic Language

Source program

Symbolic language is the notation used by the programmer to write (code) the program. The program written in SPS language is called a "source program." This language provides the programmer with mnemonic operation codes, special characters, and other necessary symbols. The use of symbolic names (labels) makes a program independent of actual machine locations. Programs and routines written in SPS language can be relocated and combined as desired. Routines within a program can be written independently with no loss of efficiency in the final program. Symbolic instructions may be added or deleted without reassigning storage addresses.

Advantages of SPS

Macro-instructions

Linkage instructions

The macro-instructions that are written in a source program are commands to the processor to generate the necessary linkage instructions. Linkage instructions provide the path to a subroutine and a return path to the user's program. These subroutines may be any of seventeen IBM Library subroutines like floating divide, square root, and arctangent; or special subroutines prepared by the user. The ability to process macro-instructions simplifies programming and further reduces the time required to write a program.

The source program is punched into an input tape; or into cards if the system is equipped with the IBM 1622 Card Read Punch unit. The source program, after it is punched, together with the Library subroutines that are required, is assembled into a finished machine language program known as the "object program." The program is self-loading (contains its own loader program) and can be run at any time.

Object program

Assembly is accomplished by the SPS processor which is available in two forms: (1) the 1620/1710 two-pass paper tape processor program, and (2) the 1620/1710 two-pass card processor program. Assembly may also be accomplished by using the 7090 processor for assembling 1620/1710 programs. Preparation of the source program is the same for all the processors. The differences between the latter processor and the 1620/1710 processors are described in appropriate sections of this manual.

SPS processors

The SPS processor programs and the Library subroutines described in this manual are available through your IBM sales representative. For distribution, the card and tape versions of the 1620/1710 SPS are numbered 1620-SP-020 and 1620-SP-021, respectively.

Function of processor programs

The function of the processor programs is to translate the symbolic language of the programming system into the language of the 1620. The translation is one for one – the processor produces one machine language instruction for each machine instruction, except macro-instructions, written in symbolic form.

Symbolic Programming

Definition

Symbolic programming may be defined as a method wherein names, characteristics of instructions, or closely related symbols are used in writing a program. The core

Operation codes

of the symbolic language is the operation code. SPS permits the programmer to write (code) in a more simple, familiar language and does not require as detailed machine knowledge because, in coding the program, the programmer uses operation codes that are in easily remembered mnemonic form rather than in the numerical language of the machine. Operation codes are of three types: Declarative, Imperative, and Control.

Declarative Operation Codes

Define areas and constants

Declarative operation codes are used for assignment of core storage for input areas, output areas, and working areas. The assigned areas are utilized by the object program and may contain the data to be processed and/or the constants (numerical or alphameric characters) required in the object program when the data is processed. Declarative statements never generate instructions in the object program, but may generate constants that are assembled as part of the object program.

Imperative Operation Codes

Instructions to the machine

Imperative operation codes specify the operations or instructions that the object program is to perform. In this group are included all arithmetic, branching, and input/output statements. Most statements on the coding sheet prepared by the programmer are of this type. These statements are translated one for one and are assembled as the machine language instructions of the object program.

Control Operation Codes

Control the processor

Control operation codes are commands to the processor that provide the programmer with control over portions of the assembly process. Instructions of this type do not normally generate instructions in the object program.

The actual and mnemonic operation codes within these categories are presented under PROGRAMMING THE 1620/1710 USING SPS.

The statements or instructions in the source program must be entered by the programmer, in logical sequence, on the coding sheet.

Coding Sheet

Source language format

The programmer enters all information relevant to the coding of the source program and subsequent assembly of the object program on coding sheet, Form X26-5627 (Figure 1). Figure 2 shows a sample input card, Form J59692. The format of the input card or paper tape record follows the headings on the coding sheet. In paper tape, the first punching position of a record is said to be column 1. The card

Field

columns assigned to a single heading are referred to as a *field.* Following is an explanation of the headings in the order of their appearance on the sheet.

Heading Line

Space is provided at the top of each page for the name of the Program, Routine, and Programmer, and for the Date. This information does not constitute part of the source program language and is not punched.

Page Number (Columns 1-2)

A 2-digit page number is entered to maintain the order of the program sheets. This normally numerical entry becomes the first two digits of each statement that is punched from the sheet.

IBM

1620/1710 Symbolic Programming System
Coding Sheet

Program: _____ Date: _____ Page No. [ ][ ] of _____
(1 2)

Routine: _____ Programmer: _____

Line	Label	Operation	Operands & Remarks
3 5	6 11	12 15	16 20 25 30 35 40 45 50 55 60 65 70 75
0,1,0			
0,2,0			
0,3,0			
0,4,0			
0,5,0			
0,6,0			
0,7,0			
0,8,0			
0,9,0			
1,0,0			
1,1,0			
1,2,0			
1,3,0			
1,4,0			
1,5,0			
1,6,0			
1,7,0			
1,8,0			
1,9,0			
2,0,0			

Figure 1. 1620/1710 SPS Coding Sheet

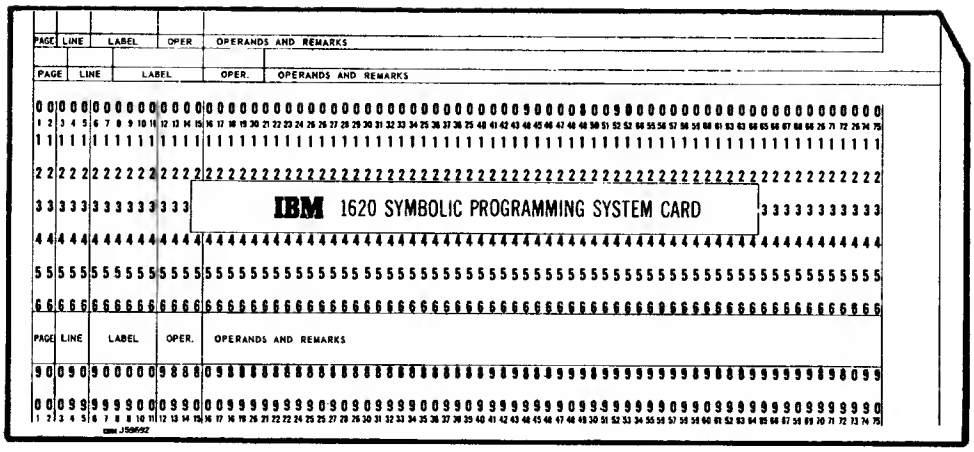


Figure 2. SPS Source Program Card

Line Number (Columns 3-5)

Sequencing source statements

A 3-digit line number is entered on the sheet to maintain the sequence of the statements coded. The first 20 lines on each sheet are prenumbered 010, 020, 030, etc., through 200. At the bottom of the sheet, six nonnumbered lines are provided for inserts or for continuing the line numbering. The inserted statement should be numbered so that it falls sequentially between the statements immediately preceding and following it. The arrangement of the prenumbered lines, 010, 020, etc., permits up to nine statements to be inserted between any two statements. After the cards for each of the lines are punched, they should be placed in correct numerical order.

Label (Columns 6-11)

The label field represents the machine location of either data or instructions. The field may be left blank or may be filled with a symbolic address. Only the data or instructions that are referred to elsewhere in the program need have a label.

Length

A label may consist of up to six alphameric characters beginning at the leftmost position in the label field. At least one of the characters must be alphabetic or one of four permissible special characters, namely, the equal sign (=), shilling symbol (/), at sign (@), and period (.).

Best labels meaningful

The best labels to select are those that are mnemonically descriptive of the area or instruction to which they are assigned. Labels that have an obvious meaning not only provide easily remembered references for the original programmer but also assist others who may assume responsibility for the program.

Operation (Columns 12-15)

Four-position field

The four-position operation field contains the actual two-digit numerical operation code or the mnemonic representation of the operation code to be performed. If the first character is numerical, no check of the operation code occurs and the numerical parts of the two-digit internal representation of the first two characters is taken as the operation code; that is, if the programmer writes 4BNF, the resulting operation code is 42. In either case, the first character of the operation code must start in the leftmost position, column 12, of the operation field. Listings of permissible mnemonic codes and actual operation codes are shown under PROGRAMMING THE 1620/1710 USING SPS.

Mnemonic or actual, left-justified

Operands and Remarks (Columns 16-75)

The operands and remarks field is used to specify the information that is to be operated upon and may contain, if desired, any additional remarks concerning the statement.

If declarative

For declarative operation statements, the first operand usually defines the length, the remaining operands, if present, specify constants, an address, and remarks.

If imperative

For imperative operation statements, the operands and remarks field contains, at most, four items: three of these are operands and the fourth, remarks. The first two operands may be the symbolic or actual addresses of data or instructions, the P and Q portions of the instruction. The third operand, which should be numerical, is called the flag indicator operand and is used to set flags in the assembled instruction. The final item consists of the remarks associated with each statement. Imperative statements need not contain all four items. Any one or more than one may be omitted. The two special characters which may not be used in an operand are the close and open parentheses,)(.

If control

A control operation statement normally consists of only one operand.

Statement Writing

Rules

Certain rules must be observed in writing or coding the statements that make up the source program. This section contains rules that apply to the statements and their elements, rules governing the length and types of statements, use of special characters, the flag indicator operand and immediate (Q) operand, types of addresses used as operands, and address adjustment by arithmetic, a method that relieves the programmer of considerable effort and reduces the number of symbols required for a source program.

Statements

Types

Symbolic statements are classed according to the operation code they contain, and thus are designated Declarative, Imperative, or Control statements. In addition to

Elements

the page and line number, a statement may contain a label, operation code, operands, and remarks.

Length

No statement in the source program may exceed 75 characters in length. Since page number, line number, label, and operation require 15 positions, the operands and remarks field may not exceed 60 characters. In the case of the paper tape SPS, the end-of-line character is considered to be part of the operands and remarks field.

Use of Special Characters in Statement Writing

The comma, asterisk, end-of-line character, blank, at (@) sign, and dollar sign are special characters which possess distinct meanings in the writing of source programs. Their use as well as that of the special characters used as operators for address adjustment are explained in detail in this section.

Comma

Use

The comma is normally used to separate items in a statement. The term *item* refers

Item

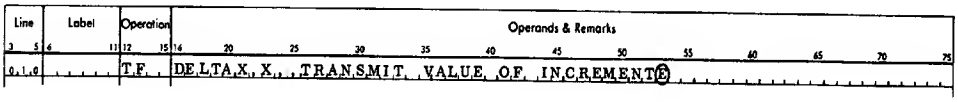
here to parts of the operands and remarks field, such as the P and Q operands, the

Items in imperative statement

flag indicator operand, remarks, length, constants, etc. An imperative statement may consist of four items: the P and Q operands, the flag indicator operand, and remarks, but need not contain all four items. Any one or more than one may be omitted.

Comma in place of omitted item

If one item is omitted and more items follow, the comma that normally follows the omitted item must be present. For example, if the flag indicator operand is omitted but remarks are present in the instruction, the format of the field will be:

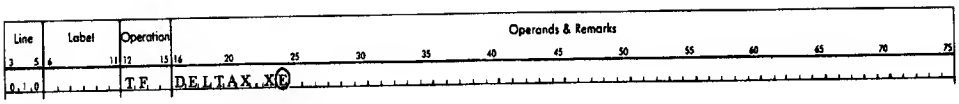


Commas before remarks

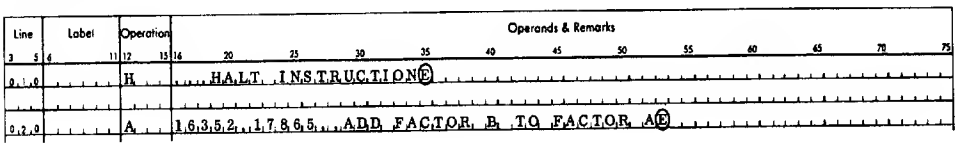
All imperative statements that contain remarks must include three commas in the operands field, even when the operands are omitted. During assembly, the omitted P or Q operands will be replaced by zeros in the P or Q portion of the assembled instruction.

No commas if last item(s) omitted

Commas indicating omission need not be present in statements in which the last item(s) is omitted. For example, in the statement in which both the flag indicator operand and remarks are omitted, no commas need be used following the second operand.



Examples



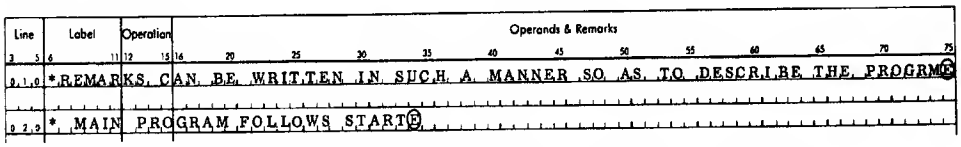
Statement 010 which is a halt operation requires three commas (, , ,) in front of the remarks, to take the place of the P, Q, and flag indicator operands. In statement 020, the first two commas set off the P and Q operands, whereas the third comma takes the place of the omitted flag indicator operand. The number of commas required for declarative statements may be one or two as explained under DECLARATIVE OPERATIONS.

Asterisk

The asterisk has three uses: in writing comments (only), as an operand or term of an operand, and in address adjustment.

Indicates comments

Lines of descriptive information may be inserted in the program by placing an asterisk (*) in column 6 of the label field. Comments then may be written in columns 7 through 75. Comments inserted in this way will appear in the symbolic output, but will not affect in any way the operation of the program. A comment statement does not produce an entry in the object program.



Statements 010 and 020 are remarks that do not generate instructions.

As first character or term(s) of an operand

The asterisk is used as the first character or *term* of an operand in an imperative statement and is interpreted by the program as the address of the high-order (leftmost) position of the address of the instruction. It may be used as any *term* of the operand to indicate the high-order (leftmost) position of the address of the instruction.

In address adjustment for multiplication

When the asterisk is used in address adjustment as an *operator*, it indicates to the processor that a multiplication must be performed in order to adjust the address.

End-of-Line Character

For paper tape

An end-of-line character (E) is required on source statements that are to be processed on paper tape. Use of this character allows statements to be located on the tape immediately adjacent to each other, with no intervening blank characters. The statements are in "free" form; that is, they are not assigned a fixed number of positions.

Not for cards

Source statements that are to be processed in punched card form do not require an end-of-line character; the remainder of the line is left blank and this is recognized by the processor as the end of the statement.

12-5-8 code

When the end-of-line character is punched in a card for off-line conversion to paper tape, it is represented by a 12, 5, 8 punch combination.

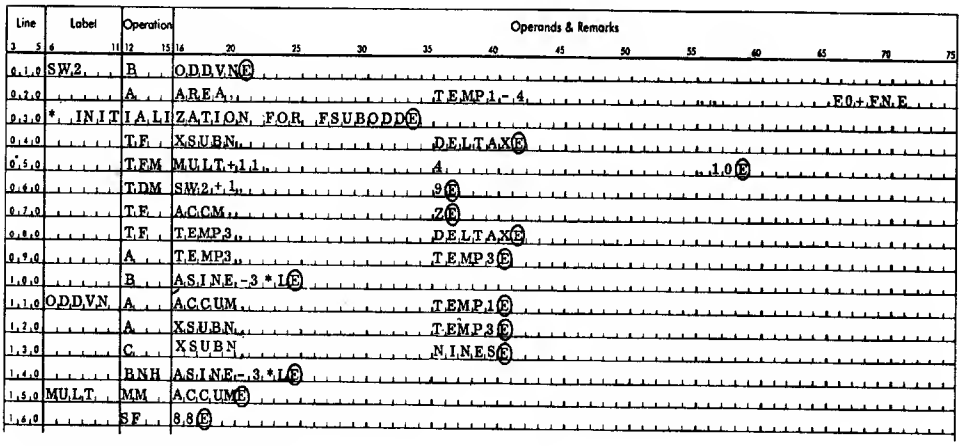
Blank Character

Ignored except in DAC statements

A blank character in operands of the source statement is ignored by the processor except in DAC statements (alphameric constants), in which blanks are considered

valid characters. In effect, the statement is condensed before it is processed.

Because blanks are ignored by the processor, the programmer, to achieve clarity on his coding sheets and output listing, may write his statements in modified "fixed" form.



In this example, columns 16, 36, and 57 are arbitrary choices for the locations of the operands. The comma following or replacing the P operand may be in any column from 16 through 35; the comma following or replacing the Q operand must be in column 56, the position preceding the flag indicator operand.

No blanks in flag indicator operand

Blanks are not permitted within a flag indicator operand. For paper tape input, this operand must begin immediately following the second comma and must be immediately followed by a comma or end-of-line character. For card input, the flag indicator operand *can* be followed by a comma, record mark, or blanks in the remainder of the card. A blank or blanks in the address operand of a *declarative* statement, when set off by commas, is interpreted by the processor as a zero address.

"At" Sign

As a constant

When the "at" sign (@) is used as part of a constant being defined by a DC, DSC, or DAC statement, a record mark (=) is created by the processor and inserted into the constant in place of the @. Specific rules for use of the @ are covered under DECLARATIVE OPERATIONS.

Dollar Sign

Signal to processor

The dollar sign ($) is used in an operand to instruct the processor that the symbolic address in an operand has a specific heading character. The $ is written between the heading character and the symbol. For example, in an operand the heading character "5" and the symbol "SUM" appear as 5$SUM. For additional information on the use of the $, refer to HEAD - HEADING in the Control Operations section.

Operands

Flag Indicator Operand

The flag indicator operand specifies the positions that are to be flagged in the assembled instruction. These positions are numbered from left to right, 0 through 11, and must be listed sequentially. For example, if positions 2, 7, and 10 are to be flagged, the flag indicator operand should be coded 2710, not 2107. All positions may be flagged, if desired. The operand then will be coded 01234567891011 and must be written in that order.

Must be sequential

Omitted flag indicator operand

Normally no flags are set when the flag indicator operand is omitted. However, if the flag indicator operand is omitted from all immediate instructions, ex-

cept TDM, a flag is automatically set in position Q_7. If the operand is present, only the positions indicated are flagged.

In indirect addressing

The flag indicator operand can be used to insert a flag over the units position of the P and Q addresses, if the source program is written for a 1620 or 1710 that has Indirect Addressing (special feature).

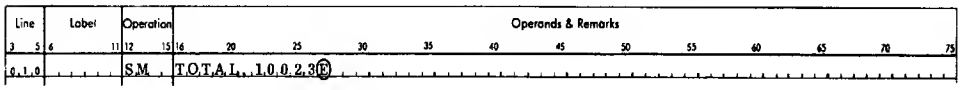
Immediate (Q Operand)

With immediate instructions

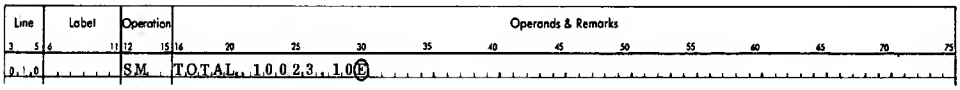
With immediate-type instructions such as Add Immediate (AM), Subtract Immediate (SM), and with actual operation codes that begin with the digit 1, the Q operand represents the actual data to be used by the instruction. It may be absolute or symbolic as previously defined. High-order zeros of absolute data may be eliminated.

Q_7 automatically flagged

During assembly, the processor automatically places a flag over position Q_7 of an immediate instruction unless a flag indicator operand indicates otherwise. For example, the statement



causes the numbers $\bar{1}0023$ to be subtracted from the field called TOTAL because the flag that terminates the field to be subtracted is automatically placed over position Q_7. However, the statement



will cause only the number $\bar{2}3$ to be subtracted from the field called TOTAL because the flag indicator operand directs that the field-terminating flag be placed over position Q_{10} rather than Q_7. There is one exception to this rule: a transmit digit immediate instruction (TDM, code 15) does not require a flag; therefore, none is automatically set by the processor.

Exception: TDM

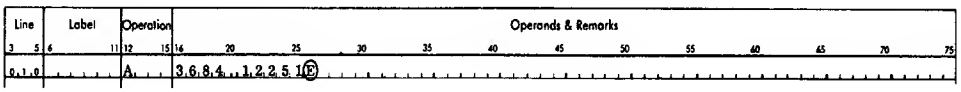
Types of Addresses Used as Operands

Operands assembled by the processor may be of three types: actual, symbolic, and asterisk. The individual applications for a particular type of address are described in the section PROGRAMMING THE 1620/1710 USING SPS.

Actual Address

Length

An actual address consists of five digits 00000-19999 for a standard capacity machine and is, as the name implies, the actual core storage address of a piece of data or an instruction. High-order zeros of an actual address may be eliminated. For example, the statement



causes the data in storage location 12251 to be added to the data in storage location 03684.

Symbolic Address

Must be defined elsewhere

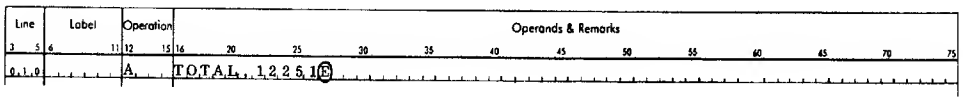
Length

A symbolic address is the name assigned by the programmer to the location of an instruction or a piece of data. A symbolic address is valid only if it is defined (given an actual numerical value) by a declarative statement somewhere in the source program or if it is used as the label of an instruction. Symbolic addresses may contain from one to six characters (letters, digits, or special characters) with the following restrictions:

a. At least one character must be nonnumerical.
b. The only permissible special characters are: equal sign (=), shilling symbol (/), at sign (@), and period (.).

It should be noted that blanks have no meaning within a symbol because they are eliminated during assembly.

The example shown below contains both an actual address and a symbolic address.

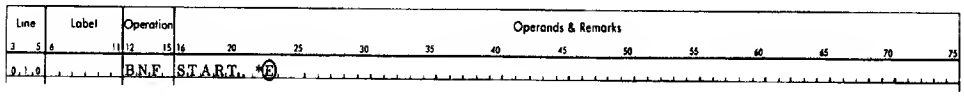


In this example, the data in the field whose *actual* address is 12251 is added to a field whose address is the *symbolic* name TOTAL.

Asterisk Address

As prefix to imperative operand

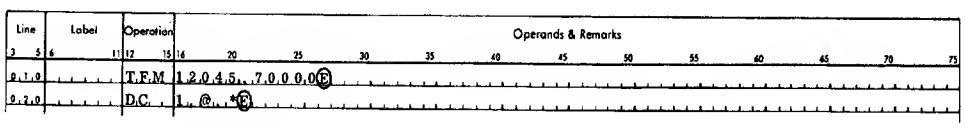
When the asterisk is used as the first character of an operand in an imperative operation, it is interpreted by the processor as meaning the address of the high-order (leftmost) position of the instruction itself. For example, the statement



indicates to the processor that the Q portion of the instruction should contain the address of the instruction itself. This instruction is assembled as 44 01234 01876 where START equals 1234 and the address assigned to the instruction is 1876. Thus, when executed in the object program, this instruction examines its own leftmost position (1876) for a flag and either branches to the instruction at location 01234 or continues, on the basis of the examination, to the next instruction located at 01888.

As prefix to declarative or control operands

When an asterisk (*) address is used with either declarative or control operations, it refers to the rightmost position of storage last assigned by the location assignment counter of the processor — not to the leftmost character of the instruction. For example, the statements



produce the instruction

16120457000≠

Since record marks can be defined only in declarative operations, an imperative statement should be followed by a DC statement when a record mark is required in the instruction. The rightmost position of the instruction is the rightmost position of storage last assigned; therefore it is also the position where the ± (constant) is stored.

Address Adjustment of Operands

Used with all types

Address adjustment is used to tell the processor to arithmetically adjust the addresses in operands. It is permitted with all types of addresses: actual, symbolic, and asterisk, and is used to refer to a location that is a given number of positions away from a specific address. Use of this feature of the language reduces the number of symbols necessary for a source program.

Math operators

By writing a + (plus sign) for addition, −(minus sign) for subtraction, and * (asterisk) for multiplication, immediately after the first or subsequent term of an operand (an asterisk as a term of an operand does not represent multiplication but means the address of the instruction, as previously explained), the programmer indicates to the processor that the address is to be adjusted.

Size of operand

Arithmetically adjusted operands may take the form of a $A \overset{*}{\pm} B \overset{*}{\pm} C \overset{*}{\pm} D$, where the terms A, B, C, and D may be numerical quantities. The number of terms in the operand is limited only by the size of the operand and remarks field. Thus the operand A + B * C − D may be further adjusted by writing after the last term another term, E, e.g., A + B * C − D + E.

Multiplication performed first

In arithmetically adjusted operands, the operation or operations of multiplication are always performed first, followed by the addition and subtraction required to calculate the adjusted address. Intermediate results that are greater than 10 digits, or a final result (adjusted address) that is over 5 digits, cannot be calculated by the processor.

Size of results

Addresses limited to storage capacity

For the 1620 with standard storage capacity (20,000 storage positions), addresses that exceed 19999 are considered errors; however, they will not be detected as such. Therefore it is possible with a standard capacity machine to assemble an object program for a machine with 40,000 or 60,000 positions of storage. For machines that have 40,000 or 60,000 positions, the processor can be modified to use the additional storage to enlarge the size of the symbol table, as explained later. In that case, addresses that do not exceed 39999 or 59999, depending upon the storage capacity, are considered valid addresses.

Caution

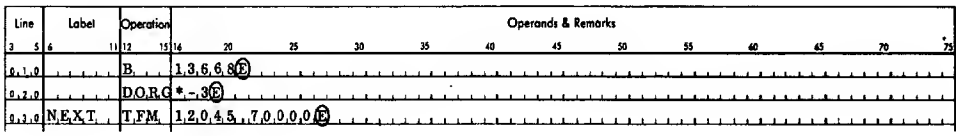
In using address adjustment, the programmer should be careful that insertions or deletions do not affect the adjusted address. For example, if a P operand in a branch (B) instruction refers to an address as * +48 (i.e., branch to the instruction that follows the next three sequentially higher instructions), the programmer must ensure that no new instructions are introduced within the three instructions to make the * +48 incorrect. In this example the asterisk (*) is the leftmost position of the instruction itself.

Examples

Line	Label	Operation		Adjusted Address		Arithmetic
010			ALPHA+40	01040	=	1000 + 40
020			ALPHA-30	00970	=	1000 - 30
030			ALPHA+2*L	01008	=	1000 + (2 x 4)
040			ALPHA*3	03000	=	1000 x 3
050			ALPHA*L	04000	=	1000 x 4
060			500+20*3-11	00549	=	500 + (20 x 3) - 11
070			100*5+20*3-11	00549	=	(100 x 5) + (20 x 3) - 11
080			*+12	02012	=	2000 + 12
090			**3*2	12000	=	(2000 x 3) x 2

The operands shown will produce the adjusted addresses, as indicated, provided the location 1000 has been assigned to the symbolic address ALPHA, the location 4 has been assigned the symbolic address L, and the instruction location (*) is equivalent to 2000.

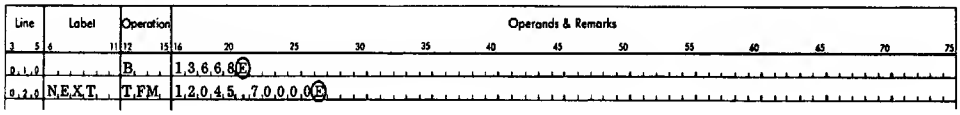
In some instructions such as the branch instruction, the Q address is not used, although a zero (00000) address is generated. Thus the instruction uses 12 storage positions. By using an * address in the following statements



the instructions are condensed, to eliminate four positions of the unused (zero) Q address, and are stored as

$$4913668016120457\bar{0}000$$

whereas the statements



are stored as

$$49136680000016120457\bar{0}000$$

To conserve storage

because the unused Q address is not eliminated. In the first example, only four positions of storage are saved; however, a considerable amount of storage can be saved in a program that contains many instructions where the Q or both the Q and P portions of instructions are unused. Because the * in the DORG statement (see CONTROL OPERATIONS) refers to the rightmost position of storage last assigned (Q_{11} of the B instruction), * —3 is the address where the next instruction starts.

Indirect Addressing

By placing a minus sign in front of the first term of an operand, a flag (minus sign) can be inserted over the units position of the adjusted address. This feature of address adjustment can be used for inserting flags required for Indirect Addressing (special feature). However, an operand written as —0 (minus zero) does not insert the flag in the units position over the zero. When the minus sign is written in front of the first term in order to set a flag over the units position, other signs following the first term should be reversed so that the correct address is obtained.

This section describes in detail the various steps to be followed in writing a program for the 1620 or 1710 using SPS. The material has been divided into three categories: Declarative Operations, Imperative Operations, and Control Operations. The imperative operations that apply to the 1710 only are described under 1710 IMPERATIVE OPERATIONS.

Declarative Operations

Assign storage areas

In programming the 1620, all records and any other data that is to be processed by the program must be assigned storage areas. Normally, all records and data to be processed consist of fields of known length and arrangement. Unless otherwise specified, areas are automatically assigned core storage locations in the order in which they appear in the source statements.

To assign addresses for instructions, constants, etc., the processor uses an address assignment counter. This counter is adjusted for each assignment made by the processor. If an address is assigned by the programmer, the counter is not adjusted.

Not executed in object program

The declarative statements provide the object program with the input/output areas, work areas, and constants it requires to accomplish its assigned task. These statements do not produce instructions that are executed in the object program. The entries, DS, DSS, DAS, and DSB assign storage. The entries, DC, DSC, DAC, DSA, and DNB usually assign storage, and also produce, in the object program, both the machine address of the area assigned and the constants that are to be stored in this area. Constants are then loaded with the object program.

Placement in source program

Declarative statements may be entered at any point in the source program. However, these statements are normally placed by themselves, preferably at the beginning or end of the program – not within the instruction area. If not placed at the beginning or end, the programmer is required to branch around an area assigned to data so the program will not attempt to execute what is in a data area as an instruction.

The declarative mnemonic operation codes and their description are as follows:

Declarative codes

Code	Description
DS	Define Symbol (Numerical)
DSS	Define Special Symbol (Numerical)
DAS	Define Alphameric Symbol
DC	Define Constant (Numerical)
DSC	Define Special Constant (Numerical)
DAC	Define Alphameric Constant
DSA	Define Symbolic Address
DSB	Define Symbolic Block
DNB	Define Numerical Blank

DS — Define Symbol (Numerical)

Assigns numerical values to labels

A DS statement may be used to define symbols used in the source program (i.e., to assign storage addresses or values to symbolic addresses or labels) and to assign storage for input, output, or working areas. A DS statement does not cause any data to be loaded with the object program.

First operand defines length

The length of the field is defined by the first operand. This operand may be an absolute value or a symbolic name. If a symbolic name is used, the symbol must previously have been defined as an absolute value, that is, it must have appeared

in the label field in a statement of the source program preceding the one in which it is used. Address adjustment may be used with this operand.

Second operand specifies storage address

The address in core storage of the field being defined may be assigned by the programmer or the programmer may let the processor assign the address. If the processor assigns the address, the statement is terminated after the first operand. If the programmer assigns the address, a second operand, which may be symbolic, asterisk, or actual, is used to establish the address of the field. Since data fields are addressed at their rightmost (low-order) digit the processor assigns this position as the address of the field. Address adjustment may be used with the second operand. If the second operand is symbolic, it also must previously have been defined. Addresses assigned by the programmer do not disrupt the sequence of addresses assigned by the processor.

Defines symbol without reserving storage

A DS statement may also be used to define a symbol, without assigning any storage, i.e., to define it as an absolute value. In this case, the first operand is omitted (or written as 0) and the second operand represents the value (may not exceed five digits in length). The second operand may be an actual value or a previously defined symbol. To define storage which will not be referred to symbolically, the label of the DS statement may be omitted.

Defines field length only

The following statements define the field length only. When remarks are added to the statement, the field length must be followed by two commas.

```
Line  Label    Operation  Operands & Remarks
0,1,9 DELTAX   DS         7
0,2,0 DELTAX   DS         7,,,TWO COMMAS ARE REQUIRED WHEN REMARKS ARE INCLUDED
```

In the next example, the programmer assigns the address of the field and excludes the field length (the first operand) from the statement because it is without significance, replacing it with a comma. The following statements cause the processor to associate the address 12930 with the label SUM:

```
Line  Label    Operation  Operands & Remarks
0,1,0 SUM      DS         ,1,2,9,3,0
0,2,0 SUM      DS         ,1,2,9,3,0,,TWO COMMAS ARE AGAIN REQUIRED
```

Again, in this example, two commas are required when remarks form part of the statement.

The following statement which is similar to the one previously given is assigned a value that is other than an address.

```
Line  Label    Operation  Operands & Remarks
0,1,0 FL       DS         ,1,7,,FIELD LENGTH USED BY SUBSEQUENT STATEMENTS
```

For defining input/output storage, numerical

This statement defines the symbol FL as being equivalent to the value 17. Subsequent uses of this symbol are permitted because the symbol has been defined.

It should be noted that an area defined by the processor for a DS statement is always addressed at the rightmost position. However, to use this area for input/output, the leftmost digit must be addressed. This is done by using a DSS statement in place of a DS statement or by address adjustment with a DS statement, which

subtracts a number that is one less than the length of the area from the address of the area. In a previous example, where DELTAX was defined as having a field length of 7, the operand of another instruction that read numerical data into the DELTAX field was written as DELTAX—6.

DSS — Define Special Symbol (Numerical)

Defines input/output storage area

The DSS statement is similar to the DS statement with one exception: when the second operand is omitted, the processor assigns the leftmost position as the address of the field. If a second operand is assigned by the programmer, this address is assumed to be equivalent to the leftmost position of the field. A DSS statement is normally used to define a storage area for input/output. The data in such an area may be moved during execution of the object program by a transmit record instruction which requires that an address assigned to an area must be that of the leftmost position.

DAS — Define Alphameric Symbol

The DAS statement is similar to the DS statement with two exceptions:

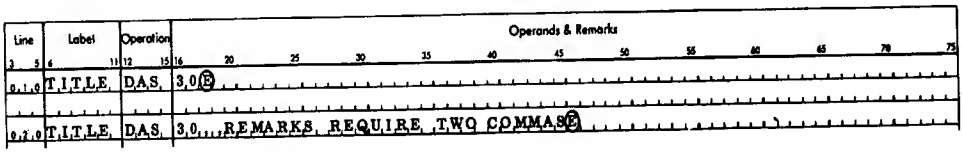
Length automatically doubled

1. The length specified by the first operand is automatically doubled by the processor to allow for alphameric data. Each alphameric character requires two storage positions.

Address always odd-numbered

2. The address of the field, if generated by the processor, is the leftmost position of the field plus one. The position is always odd-numbered, as it must be with any alphameric field.

The following example illustrates a DAS statement.



For defining input/output storage, alphameric

This statement defines an area for input/output that can contain 30 alphameric characters. The processor assigns 60 positions in core storage to accommodate alphameric coding. The output listing indicates this by typing 30 when this statement is assembled and listed. The omission of the second operand causes the processor to assign an address. During internal transmission of a field which utilizes an input/output area that is defined with a DAS, the area must be addressed at its rightmost position. In the example, the address may be achieved through address adjustment, i.e., TITLE+2*30—2.

DC — Define Constant (Numerical)

The DC statement may be used to enter numerical constants into the object program, and, for ease of reference, to assign names to the constants. The label field contains the name by which the constant is known. DC statements consist of three operands. The first operand indicates the length of the constant field; the second, the actual constant; the third, the storage address of the constant. The third operand is not used when the programmer prefers to let the processor assign the storage address. The assigned address is the rightmost storage position of the constant. The leftmost storage position is the position over which the processor places a flag.

Three operands

Three commas

Whenever remarks form part of a DC statement, three commas must be included in the statement. The first and third operands may be symbolic or actual. They are subject to address adjustment. A symbolic address must previously have been defined to be valid.

Length

If the first operand (length of constant) is smaller than the constant, an invalid condition results. If it is larger than the constant, zeros are inserted to the left of the constant so that the number of zeros plus the number of positions in the constant equals the length of the field (first operand).

Constant

A constant that is a positive number will be stored in the form of an unsigned integer; a negative number, in the form of a signed integer. A negative number has the minus sign written in front of the constant as part of the second operand. During assembly, a negative number produces a flag (minus sign) over the units position of the constant.

Sign Flag

If the constant 0100000 and –0004337769 are required, they may be defined as follows:

```
Line  Label    Operation  Operands & Remarks
0,1,0 CONST1   DC         7,,1,0,0,0,0,0,E
0,2,0 CONST1   DC         7,,1,0,0,0,0,0,,,INCLUDE 3 COMMAS FOR REMARKS@
0,3,0 CONST2   DC         1,0,,-4,3,3,7,7,6,9E
0,4,0 CONST2   DC         1,0,,-4,3,3,7,7,6,9,,,INCLUDE 3 COMMAS FOR REMARKS@
```

In both cases, constant 1 or constant 2, the length of field is greater than the constant, and the address of these constants is assigned by the processor. These constants will appear in the object program as

$$\bar{0}100000$$

$$\bar{0}004337769\bar{}$$

Record mark

A record mark may be used in a constant but must be in the units position and must be written as the character @. The following example contains statements that:

1. Store a record mark by itself as a constant.
2. Store a constant 6 and record mark.
3. Store a minus 0773 and record mark.

```
Line  Label    Operation  Operands & Remarks
0,1,0 RMARK    DC         1,,@,,,STORE A RECORD MARK ONLY@
0,2,0 CONSTX   DC         2,,6@,,,STORE A SIX AND RECORD MARK@
0,3,0 CONSTY   DC         5,,-7,7,3@,,,STORE A MINUS 773 AND RECORD MARK@
```

These constants appear in the object program as:

$$\doteq$$

$$\bar{6}\doteq$$

$$\bar{0}77\bar{3}\doteq$$

A constant 7 with a flag ($\bar{7}$) is generated by either of the following statements:

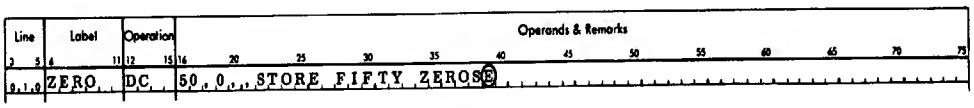
```
Line  Label    Operation  Operands & Remarks
0,1,0 CONSTZ   DC         1,,-7,,,STORE A 7 WITH A FLAGE
0,2,0 CONSTZ   DC         1,,7,,,STORE A 7WITH A FLAGE
```

One-digit constant always flagged

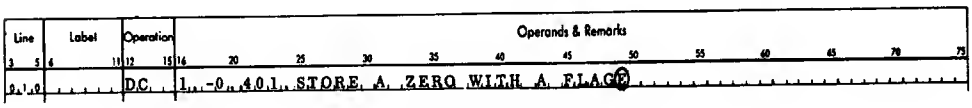
A flag is always placed over a one-digit constant, regardless of the sign (positive or negative). Therefore the programmer must use two positions to define a positive one-digit constant.

Maximum length constant

Constants may not exceed 50 characters. The following statement generates a constant containing 50 zeros.



To store a zero with a flag at location 401, the following statement can be used:



Because a label is not included in this statement, the actual address (401) must be used by any other instruction when referring to this constant.

DSC — Define Special Constant (Numerical)

The DSC statement is similar to the DC statement with two exceptions:

1. When the third operand is omitted, the address assigned by the processor to the field is that of the leftmost position of the field. If the third operand is present, the address of the constant is assumed to be the leftmost position of the field, and the constant will be stored with its leftmost digit at this address when the object program is loaded.
2. A flag is not placed in the leftmost position of the field.

DAC — Define Alphameric Constant

To define a constant consisting of alphameric data, the operation code DAC is used. The DAC statement is similar to the DC statement with three exceptions:

1. The first operand (length) is automatically doubled by the processor to allow two storage positions for each alphameric character.
2. The storage address of the constant is the address of the leftmost position plus one. This address must be an odd-numbered address to comply with the requirements for alphameric data storage. An odd-numbered address will automatically be assigned, if it is assigned by the processor. If it is specified by the programmer (as in line 020 of the following example), the processor assigns the specified address and provides that the constant is stored beginning one position to the left of the specified address. In the latter case, the processor makes no test of whether or not the address is odd-numbered or whether the address (or the position to the left) has been previously assigned.
3. High-order zeros are not automatically inserted in the constant by the processor, as is the case with a DC statement when the field length exceeds the number of characters. The number of characters including blank characters should not be greater or less than the specified length (first operand). When the rightmost position or positions of the constant are blank characters, they should be followed by a comma or end-of-line character. For card input, the rightmost position must be followed by a comma or a record mark.

NOTE: Only DAC and DNB instructions may be used to insert blank characters into storage.

Line	Label	Operation	Operands & Remarks
010	NOTE1	DAC	17, DECK 3478 PUNCHED, END OF JOB MESSAGE(E)
020		DAC	25, ,901, STORE ALPHA BLANKS 900-949(E)
030	RMARK	DAC	1,@, ALPHA RECORD MARK FOR OUTPUT AREA(E)
040	CONST	DAC	30, FOR DELTAX=0,.000, AREA=0, .00000@, STORE CONSTANT, RECMK(E)

In the example shown:

1. Statement 010 uses 34 storage positions to store the 17-position constant (deck 3478 punched).
2. Statement 020 places 25 alphameric blanks into storage locations 900 through 949. Also, a flag is set in location 900.
3. Statement 030 records an alphameric record mark in storage.
4. Statement 040 places a 30-position constant, including a record mark, in storage. The second comma in this statement is part of the constant and the fifth comma is part of the remarks.

Address adjustment

A 50-character alphameric constant (maximum allowable) occupies 100 positions of storage. A flag is set over the leftmost position of the field. Addressing this constant for internal field transmission requires the address OUTPUT +50 * 2—2, where OUTPUT is the symbol (label) which represents the leftmost address plus one.

DSA — Define Symbolic Address

Maximum, ten addresses

The DSA statement may be used to store a series of up to ten addresses as constants, as part of the object program. These addresses can be used for instruction modification or for setting up a table of addresses through which the programmer may index to modify a routine.

Restrictions

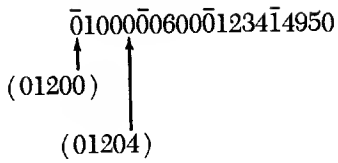
Each entry (symbolic or actual) in the operands field, with the exception of the last entry, is followed by a comma. The equivalent machine address of each entry is stored as a 5-digit constant. The constants are stored adjacent to each other with a flag over the high-order position of each. The label field of this statement must contain the symbolic name by which the table of constants may be referred to. An address at which this table is stored in core storage may not be assigned by the programmer nor may any remarks be included in the DSA statement. The address assigned by the processor is the address at which the rightmost digit of the first constant will be located.

NOTE: If the last operand is followed by a comma, an additional zero address ($\bar{0}0000$) is assembled in the table.

In the example that follows, symbols ALPHA, ORIGIN, and OUTPUT are equivalent to address 1000, 600, and 15000, respectively.

Line	Label	Operation	Operands & Remarks
010	TABLE	DSA	ALPHA, ORIGIN, 1234, OUTPUT-50(E)

The constants are stored as



If the leftmost digit of these four constants is located at 01200, then the address equivalent to TABLE will be 01204, the location of the rightmost digit of ALPHA.

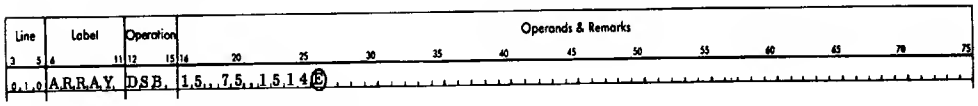
DSB — Define Symbolic Block

For storing numerical array

A DSB statement is used to define an area of storage for storing a numerical array. A DSB statement does not cause any data to be loaded with the object program. The label of this statement is converted to the address at which the first element of the array is stored (i.e., the rightmost position of the first element). The first operand indicates the size of each element; the second, the number of elements.

Actual or symbolic

Either or both operands may be symbolic or actual. If symbolic, the symbol must have been previously defined. A third operand is required if the programmer wishes to assign the address. For example, to store an array of 75 elements, each element containing 15 digits, the statement used would be:



In this example, the array begins at location 01500 (leftmost position of the first 15-digit element). ARRAY is equivalent to 01514 (address of the first element).

DNB — Define Numerical Blank

Maximum, fifty blanks

A DNB statement is used to define a field of numerical blanks. (The 8-4 card code denotes a numerical blank.) Up to fifty blanks may be specified in each DNB statement. In addition to a label, two operands can be assigned by the programmer. The first of these specifies the number of blank characters desired (field length) and the second, the rightmost address of the field where the blanks are stored in the object program.

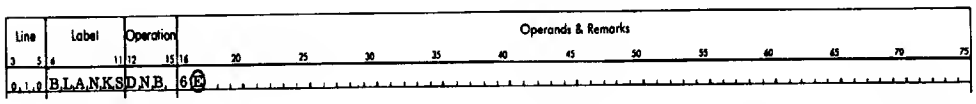
No flag in leftmost position

If the second operand is omitted and the statement is labeled, the address assigned to the label by the processor is the rightmost storage position of the blank field. The blank field does not contain a flag in its leftmost position.

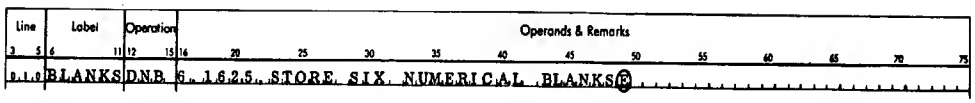
Programmer provides flag or record mark

If the programmer wishes to move a blank field in core storage, he must either define a single-digit constant with flag bit in the position in front of the leftmost position of the blank field or a record mark in the position following the rightmost position of the blank field.

If six numerical blanks are required, they may be defined as follows:



The processor assigns the storage address of the six blank positions to the label BLANKS. In the example that follows, the programmer assigns the storage address as 01625.



Two commas required

In a DNB statement, two commas are required whenever remarks are included in the statement; the first after the length operand and the second after, or in place of, the address operand.

Summary of Declarative Operations

Location assignment counter

As stated earlier, areas being defined by the processor are assigned core storage locations in the order in which they are processed. To do this, the processor program uses a location assignment counter to keep track of the address of the last assigned storage location. Table 1 shows the amount added to the location assignment counter for each instruction and summarizes the coding and operation of

Table 1. Summary of Declarative Operations

NOTE: Except for the constants in DC, DSC, and DAC, all operands may be
1. actual.
2. symbolic. Symbols must be previously defined except in DSA.

All operands may use address adjustment. Remarks may follow the other operands except in DSA statements.

DECLARATIVE STATEMENT FORMAT			AMOUNT ADDED TO LOCATION ASSIGNMENT COUNTER IF ADDRESS (A) IS BLANK	VALUE STORED IN SYMBOL TABLE AS EQUIVALENT TO "SYMBOL"	DATA FIELDS WHICH ARE LOADED AS A PART OF THE OBJECT PROGRAM
LABEL	OP CODE	OPERANDS			
SYM	DS	L, A (E)	L (length) If L is blank, 0 is added.	A address. If A is blank, the field address from the location assignment counter is stored.	None
SYM	DSS	L, A (E)	L (length) If L is blank, 0 is added.	A address. If A is blank, the numerical record address from the location assignment counter is stored.	None
SYM	DAS	L, A (E)	2 x L is added. If L is blank, 0 is added.	A address must be odd. If A is blank, the alpha record address from the location assignment counter is stored.	None
SYM	DC	L, C, A (E)	L is added.	A address. If A is blank, the field address from the location assignment counter is stored.	C, the (numerical) constant
SYM	DSC	L, C, A (E)	L is added.	A address. If A is blank, the numerical record address from the location assignment counter is stored.	C, the (numerical) constant
SYM	DAC	L, C, A (E)	2 x L is added.	A address must be odd. If A is blank, the alpha record address from the location assignment counter is stored.	C, the (alphameric) constant
SYM	DSA	D, E, F, G, H, I, J, K, L, M (E)	5 x (number of addresses) is added.	Field address of the first address on list.	A list of the actual addresses that correspond to D, E, F, etc.
SYM	DSB	L, N, A (E)	Length of each element x number of elements is added.	A address. If A is blank, field address of the first element is stored.	None
SYM	DNB	L, A (E)	L is added.	A address. If A is blank, the field address from the location assignment counter is stored.	Number of blank characters that equal L.

Alpha record address
Field address
Numerical record address

each declarative operation. "Alpha Record Address" in the table refers to the leftmost position plus one of an alphameric field, whereas "Field Address" refers to the rightmost position of a field. The term "Numerical Record Address" refers to the leftmost position of a field.

Imperative Operations

This section describes the operations (instructions) written in symbolic language that are translated by the processor into 1620 machine language. The function of each machine language operation code is discussed in the *1620 Reference Manual* (Form A26-4500) and the *1710 Reference Manual* (Form A26-5601).

Imperative operations may be divided into five classes:

1. Arithmetic
2. Internal data transmission
3. Branch
4. Input/Output
5. Miscellaneous

In this section are presented thirty-two standard 1620 instructions and fifteen special feature instructions that compose the imperative operations. For each class, a listing of the instructions shows the actual operation code, the mnemonic representation for SPS, the P and Q address requirements and function, and the addresses that can be modified using Indirect Addressing (special feature). The flag indicator operand can be used to insert a flag over the units position of an address (P or Q) when this special feature is used.

Summary of statement writing rules

As stated earlier concerning coding sheet fields:

1. Any instruction in the source program may be labeled.
2. Mnemonic or actual operation codes must be recorded for each statement.
3. For each instruction, up to four items can be entered in the operand and remarks field: P operand, Q operand, flag indicator operand, and remarks.
4. Commas must separate these items or be used in place of omitted items.
5. Statements written for the card processor do not require an end-of-line character (E).
6. Operands may be symbolic or actual, and are subject to address adjustment.

Arithmetic Instructions

Listed in Table 2

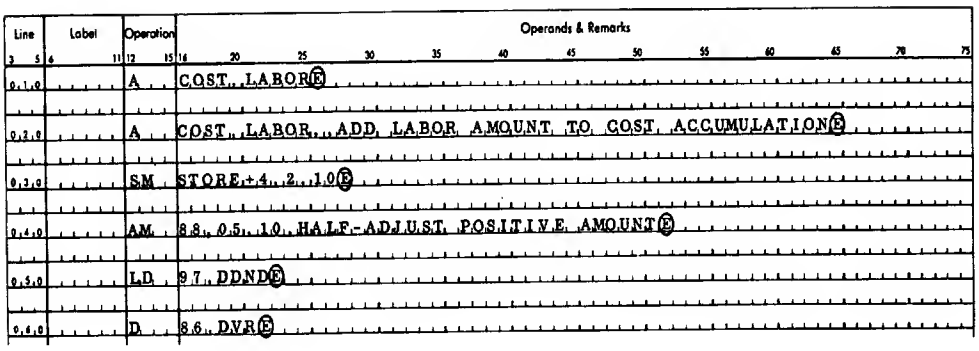
Table 2 lists arithmetic instructions, eight of which are special feature instructions. Each instruction shown must have both a P and a Q address. Note that immediate-type arithmetic instructions have only a Q part and are not subject to Indirect Addressing (special feature).

Table 2. Arithmetic Instructions

NOTE: Indirect Addressing (special feature) is allowable with all P address operands listed below. An X to the right of the Q operand indicates that this feature may be used with it.

OPERATION	OPERATION CODE		OPERANDS		
	MNEMONIC	ACTUAL	P ADDRESS	Q ADDRESS	
Add	A	21	Storage address of units position of augend	Storage address of units position of addend	X
Add Immediate	AM	11	Same as code 21	Q_{11} of instruction is units position of addend	
Subtract	S	22	Storage address of units position of minuend	Storage address of units position of subtrahend	X
Subtract Immediate	SM	12	Same as code 22	Q_{11} of instruction is units position of subtrahend	
Multiply	M	23	Storage address of units position of multiplicand	Storage address of units position of multiplier	X
Multiply Immediate	MM	13	Same as code 23	Q_{11} of instruction is units position of multiplier	
Load Dividend (special feature)	LD	28	Storage address in product area to which units position of field (dividend) is to be transmitted	Storage address of units position of dividend	X
Load Dividend Immediate (special feature)	LDM	18	Same as code 28	Q_{11} of instruction is units position of dividend	
Divide (special feature)	D	29	Storage address at which first subtraction of the divisor occurs	Storage address of units position of divisor	X
Divide Immediate (special feature)	DM	19	Same as code 29	Q_{11} of instruction is units position of divisor	
Floating Add (special feature)	FADD	01	Storage address of units position of exponent of augend	Storage address of units position of exponent of addend	X
Floating Subtract (special feature)	FSUB	02	Storage address of units position of exponent of minuend	Storage address of units position of exponent of subtrahend	X
Floating Multiply (special feature)	FMUL	03	Storage address of units position of exponent of multiplicand	Storage address of units position of exponent of multiplier	X
Floating Divide (special feature)	FDIV	09	Storage address of units position of exponent of dividend	Storage address of units position of exponent of divisor.	X

Examples



These statements cause the following operations to be performed:

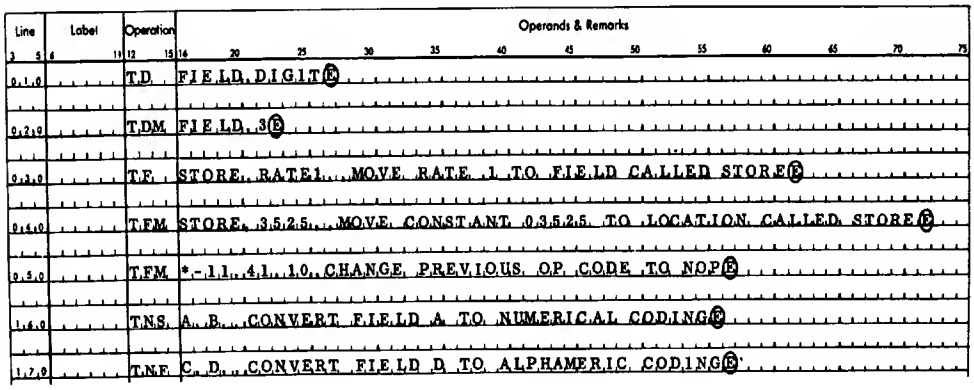
Line 010 — Add labor amount to cost amount.
020 — Same as line 010 except three commas are required for remarks.
030 — Subtract a constant 02 from the field located at STORE PLUS 4.
040 — Add a constant 05 to the field at storage location 00088.
050 — Move DDND (dividend) to the product area (storage location 00097).
060 — Divide the dividend by successive subtractions of the DVR (divisor), starting in storage location 00086.

Internal Data Transmission Instructions

Listed in Table 3

Internal data transmission instructions require both P and Q addresses. The five internal data transmission instructions that are not standard on the 1620 are: Transfer Numerical Strip (TNS), Transfer Numerical Fill (TNF), Floating Shift Right (FSR), Floating Shift Left (FSL), and Transmit Floating Fields (TFL). Table 3 lists the Internal Data Transmission instructions.

Examples



These statements cause the following instructions to be executed:

Line 010 — A numerical digit at the location called DIGIT is moved to the location called FIELD.
020 — A digit 3 is moved to the location called FIELD.

030 – Rate 1 is moved to the field called STORE.
040 – A constant 3525 is moved to the location called STORE.
050 – A constant 41 is moved to O_0 and O_1 positions of the preceding instruction in the object program.
060 – Field A is moved to field B and converted from alphameric coding (2 digits per character) to numerical coding (1 digit per character).
070 – Field D is moved to field C and converted from numerical coding to alphameric coding.

Table 3. Internal Data Transmission Instructions

NOTE: Indirect Addressing (special feature) is allowable with all P address operands listed below. An X to the right of the Q address operand indicates that this feature may be used with it.

OPERATION	OPERATION CODES		OPERANDS		
	MNEMONIC	ACTUAL	P ADDRESS	Q ADDRESS	
Transmit Digit	TD	25	Storage address to which single digit is transmitted	Storage address of single digit to be transmitted	X
Transmit Digit Immediate	TDM	15	Same as code 25	Q_{11} of instruction is the single digit to be transmitted	
Transmit Field	TF	26	Storage address to which units position of field is transmitted	Storage address of units position of field to be transmitted	X
Transmit Field Immediate	TFM	16	Same as code 26	Q_{11} of instruction is the units position of the field to be transmitted	
Transmit Record	TR	31	Storage address to which high-order position of record is transmitted	Storage address of high-order position of the record to be transmitted	X
Transfer Numerical Strip (special feature)	TNS	72	Storage address of rightmost position of alphameric field to be transmitted	Storage address of the units position of the numerical field	X
Transfer Numerical Fill (special feature)	TNF	73	Storage address of rightmost position of alphameric field	Storage address of the units position of the numerical field to be transmitted	X
Floating Shift Right (special feature)	FSR	08	Storage address to which units (rightmost) digit of mantissa is transmitted	Storage address (rightmost) digit of mantissa to be transmitted	X
Floating Shift Left (special feature)	FSL	05	Storage address to which high-order digit of the mantissa is transmitted	Storage address of low-order digit of mantissa to be transmitted	X
Transmit Floating (special feature)	TFL	06	Storage address to which units position of exponent is transmitted	Storage address of units position of exponent of field to be transmitted	X

Branch Instructions

Listed in Table 4

Table 4 lists the branch instructions. Note that both the BI (Branch Indicator) and BNI (Branch No Indicator) instructions require one of the fourteen switch or indicator codes listed in Table 5 as a Q address. The code indicates the switch or indi-

cator to be interrogated for status. To relieve the programmer of having to code a Q address, thirteen unique mnemonics that represent ten of the possible fifteen codes are included in SPS language for both BI- and BNI-type instructions. For a unique mnemonic, the processor generates the actual machine language code 46 (Branch Indicator) or 47 (Branch No Indicator) and the Q address that represents the switch or indicator.

Table 4. Logic (Branch and Compare) Instructions.

NOTE: Indirect Addressing (special feature) is allowable with all P address operands listed below except Branch Back. An X to the right of the Q address operand indicates that this feature may be used with it.

OPERATION	OPERATION CODES		OPERANDS		
	MNEMONIC	ACTUAL	P ADDRESS	Q ADDRESS	
Compare	C	24	Storage address of units position of the field to which another field is compared	Storage address of units position of the field to be compared with the field at the P address	X
Compare Immediate	CM	14	Same as code 24	Q_{11} of instruction is units position of the field to be compared with the field at the P address	
Branch	B	49	Storage address of the leftmost digit of the next instruction to be executed	Not used	
Branch No Flag	BNF	44	Storage address of the leftmost digit of next instruction to be executed if branch occurs	Storage address to be interrogated for presence of a flag bit	X
Branch No Record Mark	BNR	45	Same as code 44	Storage address to be interrogated for presence of a record mark character	X
Branch on Digit	BD	43	Same as code 44	Storage address to be interrogated for a digit other than zero	X
Branch Indicator	BI	46	Storage address of leftmost position of next instruction to be executed if indicator tested is on	Q_8 and Q_9 of instruction specify the program switch or indicator to be interrogated (see Table 5)	
Unique Branch Indicator mnemonics:					
Branch High	BH	None	Same as code 46	None required	
Branch Positive	BP	None	Same as code 46	None required	
Branch Equal	BE	None	Same as code 46	None required	
Branch Zero	BZ	None	Same as code 46	None required	
Branch Overflow	BV	None	Same as code 46	None required	
Branch Any Data Check	BA	None	Same as code 46	None required	
Branch Not Low	BNL	None	Same as code 46	None required	
Branch Not Negative	BNN	None	Same as code 46	None required	
Branch Console Switch 1 on	BC1	None	Same as code 46	None required	
Branch Console Switch 2 on	BC2	None	Same as code 46	None required	

Table 4. Logic (Branch and Compare) Instructions (Contd.)

OPERATION	OPERATION CODES		OPERANDS	
	MNEMONIC	ACTUAL	P ADDRESS	Q ADDRESS
Branch Console Switch 3 on	BC3	None	Same as code 46	None required
Branch Console Switch 4 on	BC4	None	Same as code 46	None required
Branch Last Card (special feature)	BLC	None	Same as code 46	None required
Branch Exponent Check (special feature)	BXV	None	Same as code 46	None required
Branch No Indicator	BNI	47	Storage address of leftmost position of next instruction to be executed if indicator tested is off	Q_s and Q_v of instruction specify program switch or indicator to be interrogated (see Table 5)
Unique Branch No Indicator mnemonics:				
Branch Not High	BNH	None	Same as code 47	None required
Branch Not Positive	BNP	None	Same as code 47	None required
Branch Not Equal	BNE	None	Same as code 47	None required
Branch Not Zero	BNZ	None	Same as code 47	None required
Branch No Overflow	BNV	None	Same as code 47	None required
Branch Not Any Data Check	BNA	None	Same as code 47	None required
Branch Low	BL	None	Same as code 47	None required
Branch Negative	BN	None	Same as code 47	None required
Branch Console Switch 1 off	BNC1	None	Same as code 47	None required
Branch Console Switch 2 off	BNC2	None	Same as code 47	None required
Branch Console Switch 3 off	BNC3	None	Same as code 47	None required
Branch Console Switch 4 off	BNC4	None	Same as code 47	None required
Branch Not Last Card (special feature)	BNLC	None	Same as code 47	None required

Table 4. Logic (Branch and Compare) Instructions (Contd.)

OPERATION	OPERATION CODES		OPERANDS	
	MNEMONIC	ACTUAL	P ADDRESS	Q ADDRESS
Branch Not Exponent Check (special feature)	BNXV	None	Same as code 47	None required
Branch and Transmit	BT	27	P address minus one is the storage address to which the units position of the Q field is transmitted. P address is leftmost digit of the next instruction to be executed	Storage address of units position of the field to be transmitted X
Branch and Transmit Immediate	BTM	17	Same as code 27	Q_{11} of instruction is units position of field to be transmitted
Branch Back	BB	42	Not used	Not used
Branch and Transmit Floating	BTFL	07	P address minus one is the storage address to which the units position of the exponent portion of the Q field is transmitted. P is the storage address of the leftmost digit of the next instruction to be executed	Storage address of units position of exponent of field to be transmitted X

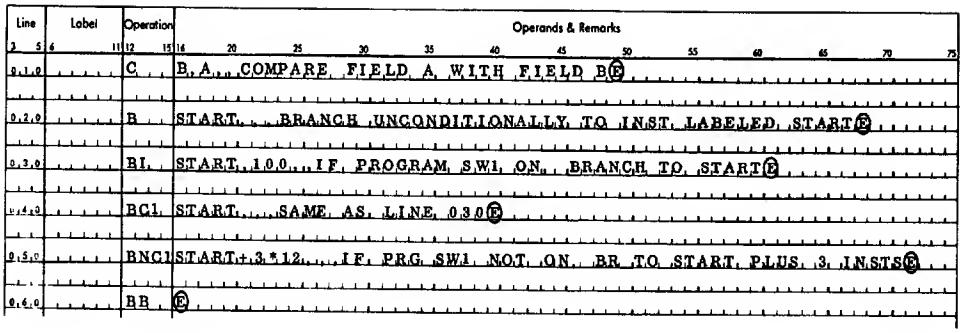
Table 5. Switch and Indicator Codes Used as Actual Q Addresses in BI and BNI Instructions.

NOTE: The additional codes for the 1710 only are shown in Table 10.

Q ADDRESS					SWITCH OR INDICATOR
Q_7	Q_8	Q_9	Q_{10}	Q_{11}	
X	0	1	Y	Z	Program Switch 1
X	0	2	Y	Z	Program Switch 2
X	0	3	Y	Z	Program Switch 3
X	0	4	Y	Z	Program Switch 4
X	0	6	Y	Z	Read Check Indicator*
X	0	7	Y	Z	Write Check Indicator*
X	0	9	Y	Z	Last Card Indicator (special feature)
X	1	1	Y	Z	High/Positive Indicator
X	1	2	Y	Z	Equal/Zero Indicator
X	1	3	Y	Z	High/Positive or Equal/Zero Indicator
X	1	4	Y	Z	Overflow Check Indicator
X	1	5	Y	Z	Exponent Check Indicator
X	1	6	Y	Z	MBR-Even Check Indicator*
X	1	7	Y	Z	MBR-Odd Check Indicator*
X	1	9	Y	Z	Any Data Check

X indicates any digit value or blank is permissible.
Y indicates any digit value is permissible.
Z indicates any digit value or blank is permissible for 1620 but digits zero and one must be excluded for 1710.
* indicates the Any Data Check indicator (19) also is on when this indicator is on.

Examples



These statements cause the following operations to be performed in the object program, as follows:

Line 010 – Compare field A with field B.
020 – Branch to an instruction labeled START.
030 – If program switch 1 is on, branch to the instruction labeled START.
040 – Same as line 030 with the exception that the unique mnemonic operation code used does not require a Q address.
050 – If program switch 1 is not on, branch to the third instruction following the one labeled START.
060 – Branch unconditionally to an instruction whose address is saved in IR-2 or PR-1.

Input and Output Instructions

Listed in Table 8

The Q operand of input and output instructions is used to specify the input/output device. Table 6 shows the five different input/output device codes that can be used as the Q address. For programming ease, SPS language includes unique mnemonic operation codes that can be used without a Q operand.

Table 6. Input and Output Device Codes Used as Actual Q Addresses with RN, WN, DN, RA, and WA Instructions.

Q ADDRESS					DEVICE
Q_7	Q_8	Q_9	Q_{10}	Q_{11}	
X	0	1	Y	Y	Typewriter
X	0	2	Y	Y	Paper Tape Punch
X	0	3	Y	Y	Paper Tape Reader
X	0	4	Y	Y	Card Punch
X	0	5	Y	Y	Card Reader

X indicates any digit value or blank is permissible.
Y indicates any digit value is permissible.

Table 7 provides the Q address for a K instruction (control operation). This address, in addition to specifying the output device (typewriter), specifies typewriter action: space, return carriage, or tabulate.

Table 7. Typewriter Control Codes Used as Actual Q Address for a K instruction.

Q ADDRESS					ACTION INITIATED
Q_7	Q_8	Q_9	Q_{10}	Q_{11}	
X	0	1	Y	1	Space
X	0	1	Y	2	Return Carriage
X	0	1	Y	8	Tabulate

X indicates any digit value or blank is permissible.
Y indicates any digit value is permissible.

Table 8 lists the five actual and mnemonic input and output operation codes and the thirty associated unique mnemonics. When a unique mnemonic is used, the processor generates the Q address of the assembled instruction.

Examples

```
Line  Label  Operation  Operands & Remarks
0,1,0        WA         OUTPUT,,100
0,2,0        WATY       OUTPUT,,,,SAME AS LINE 0,1,0
0,3,0        K          ,,101,,,,SAME AS LINE 0,4,0
0,4,0        SPTY
```

These statements cause the following operations to be performed in the object program, as follows:

Line 010 — Type out alphameric data from a storage location called OUTPUT.

020 — Same as line 010; however, a unique mnemonic is used.

030 — Single space on the typewriter.

040 — Same as line 030; however, a unique mnemonic is used.

Table 8. Input and Output Instructions

NOTE: Indirect Addressing (special feature) is allowable with all P address operands, where a P operand is required. None of the Q operands shown may be used with Indirect Addressing.

OPERATION	OPERATION CODE		OPERANDS	
	MNEMONIC	ACTUAL	P ADDRESS	Q ADDRESS
Read Numerically	RN	36	Storage address at which leftmost (first) numerical character is stored	Q_8 and Q_9 instructions specify input device (see Table 6)
Unique Read Numerically mnemonics:				
Read Numerically Typewriter	RNTY	None	Same as code 36	None required
Read Numerically Paper Tape	RNPT	None	Same as code 36	None required
Read Numerically Card (special feature)	RNCD	None	Same as code 36	None required
Write Numerically	WN	38	Storage address from which leftmost (first) numerical character is written	Q_8 and Q_9 of instruction specify output device (see Table 6)
Unique Write Numerically mnemonics:				
Write Numerically Typewriter	WNTY	None	Same as code 38	None required
Write Numerically Paper Tape	WNPT	None	Same as code 38	None required
Write Numerically Card (special feature)	WNCD	None	Same as code 38	None required
Dump Numerically	DN	35	Same as code 38	Same as code 38
Unique Dump Numerically mnemonics:				
Dump Numerically Typewriter	DNTY	None	Same as code 38	None required
Dump Numerically Paper Tape	DNPT	None	Same as code 38	None required
Dump Numerically Card (special feature)	DNCD	None	Same as code 38	None required

Table 8. Input and Output Instructions (Contd.)

OPERATION	OPERATION CODES		OPERANDS	
	MNEMONIC	ACTUAL	P ADDRESS	Q ADDRESS
Read Alphamerically	RA	37	Storage addresses at which numerical digit of leftmost (first) character is stored. (Zone digit of first character is at P minus one)	Q_8 and Q_9 of instruction specify input device (see Table 6)
Unique Read Alphamerically mnemonics:				
Read Alphamerically Typewriter	RATY	None	Same as code 37	None required
Read Alphamerically Paper Tape	RAPT	None	Same as code 37	None required
Read Alphamerically Card (special feature)	RACD	None	Same as code 37	None required
Write Alphamerically	WA	39	Storage address of numerical digit of leftmost (first) character to be written. (Zone digit of first character is at P minus one)	Q_8 and Q_9 of instructions specify output device (see Table 6)
Unique Write Alphamerically mnemonics:				
Write Alphamerically Typewriter	WATY	None	Same as code 39	None required
Write Alphamerically Paper Tape	WAPT	None	Same as code 39	None required
Write Alphamerically Card (special feature)	WACD	None	Same as code 39	None required
Control	K	34	Not used	Q_8 and Q_9 specify Input/Output device. Q_{11} specifies control functions (see Table 7)
Unique Control mnemonics:				
Tabulate Typewriter	TBTY	None	Not used	None required
Return Carriage Typewriter	RCTY	None	Not used	None required
Space Typewriter	SPTY	None	Not used	None required

Miscellaneous Instructions

Listed in Table 9

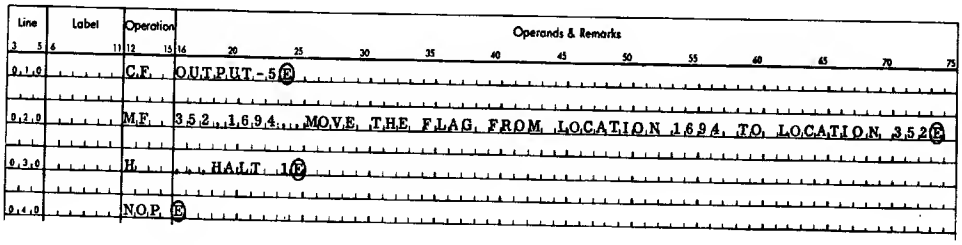
Five instructions are included in this group; one of these is a special feature, Move Flag (MF). Table 9 lists the uses of P and Q addresses in miscellaneous instructions.

Table 9. Miscellaneous Instructions

NOTE: Indirect Addressing (special feature) is allowable with all P or Q address operands that are marked with an X.

OPERATION	OPERATION CODE		OPERANDS	
	MNEMONIC	ACTUAL	P ADDRESS	Q ADDRESS
Set Flag	SF	32	Storage address at which flag bit is placed X	Not used
Clear Flag	CF	33	Storage address from which flag bit is cleared X	Not used
Move Flag (special feature)	MF	71	Storage address to which flag bit is moved X	Storage address of flag bit to be moved X
Halt	H	48	Not used	Not used
No Operation	NOP	41	Not used	Not used

Examples



These statements cause four different operations to be performed in the object program, as follows:

Line 010 – Clear a flag at the storage location, OUTPUT minus 5.
020 – Move a flag from storage location 1694 to storage location 352.
030 – Cause the program to halt.
040 – Perform no operation but proceed to the next sequential instruction.

Control Operations

The SPS language includes the following six control operations:

DORG	Define Origin
DEND	Define End
SEND	Special End
HEAD	Heading
TCD	Transfer Control and Load
TRA	Transfer to Return Address

Orders to the processor

These operation codes are orders to the processor that give the programmer control over portions of the assembly process. Specifically, DORG gives the programmer

control over the placement of his program in storage. DEND, TCD, and TRA order the processor to produce unconditional branches to locations specified by the programmer. HEAD assigns unique characters to labels or symbols used within a source program.

None labeled

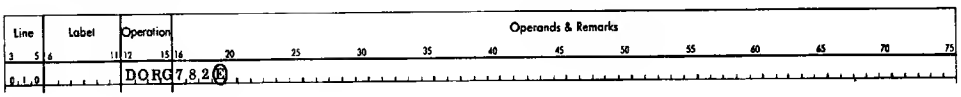
With the exception of the TRA and DORG, none of the above operations may be labeled.

DORG — Define ORiGin

Overrides automatic storage assignment

The DORG statement instructs the processor to override its automatic assignment of storage and to begin the assignment of succeeding entries at the particular location specified by the programmer. In this way the programmer is able to control assignment of storage to instructions, constants, and data. If a define origin statement is not the first entry in a source program, the processor begins the assignment of storage at location 402.

A define origin statement is coded as follows:



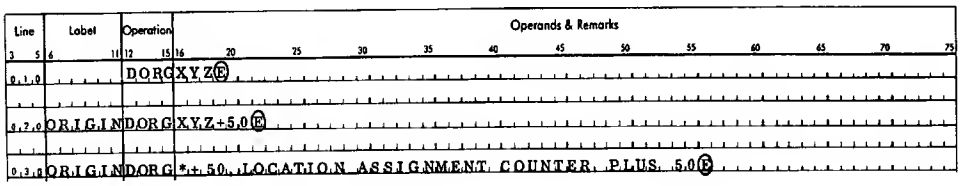
Resets location assignment counter

This statement directs the processor to reset its location assignment counter to the particular address specified in the operand (actual or symbolic), and this causes the assignment of succeeding entries to begin at this address. When an actual address is entered by the programmer, care must be taken to avoid inadvertent overlapping with areas assigned by the processor.

Overlapping in storage
Pre-empted storage area

If the operand is left blank, assignment of storage starts with 00000 address. Since the arithmetic tables are stored in locations 00100 through 00401, constants and instructions cannot occupy these storage locations.

If a symbolic address is entered, it must appear as a label earlier in the program sequence. An * address refers to the current contents of the location assignment counter. A define origin statement can take any of the following individual forms:



Examples

If XYZ (label) is previously defined as 1002, the first entry directs the processor to begin the assignment of succeeding entries at location 1002. The second entry directs the processor to begin the assignment of succeeding entries at the location that has been assigned to the symbol XYZ plus 50. The symbol ORIGIN can be used at any point in the program to refer to that address. The third entry directs the processor to begin the assignment of succeeding entries at the address specified by the current contents of the location assignment counter plus 50. A comma must follow the operand when remarks are included in a DORG statement.

DEND — Define END

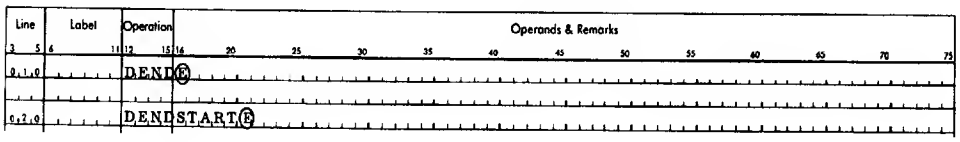
Last statement

The DEND statement is the last statement entered in the source program; it instructs the processor that all statements of the source program have been processed. A DEND

Halts after loading

statement may also be used to cause execution of the object program to begin immediately after it has been loaded. To do this, the DEND statement requires the presence of an operand representing the starting address of the program. The operand may be actual or symbolic. The 1620 will halt at the completion of loading of the object program, and execution then will begin at the address corresponding to the starting address, upon depression of the start key. If the operand specifying the starting address is omitted, the program will halt and the operator will have to start the program manually.

The following statements illustrate both types of entries.



The program is halted after *loading* by either statement. In the second case, execution begins at the address corresponding to START, upon depression of the start key.

When a DEND statement includes comments but no operand, the operand must be replaced by a comma.

SEND — Special END

Halts tape processor

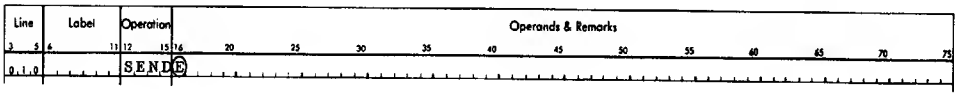
A SEND statement is provided to halt the tape processor on both passes of the source program. If a SEND statement is encountered by the card processor, the card processor will not be halted. This statement does not produce any output in the object program.

The SEND statement is used:

1. To halt the processor after one tape of a source program is processed and to allow the remainder of the source program from another tape to be threaded and then processed.
2. To halt the processor so that program switch settings may be changed and processing resumed.

No operands

The SEND statement takes the following form and requires no operands.

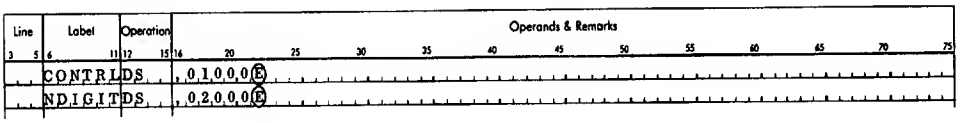


Example

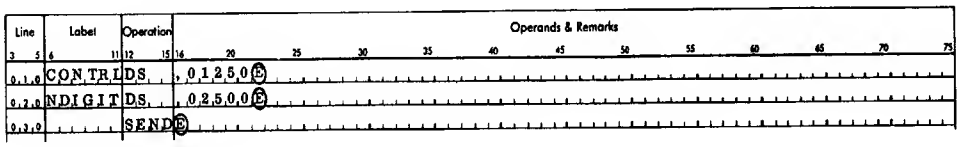
When this statement in the source program is encountered by the processor, the program halts and the message "LOAD NEXT TAPE" is typed. The operator threads the next tape or changes switch settings or both, and then depresses the start key.

If the first part of the source program is entered from the typewriter, program switch 1 will be off and the statements will be entered one at a time. When the SEND statement is entered, the processor halts. The operator may then turn on program switch 1, thread the source program tape, and continue processing the source program. SEND is especially useful where a long source program is punched in tape and certain addresses associated with labels being defined by that tape must be changed. For example, the following statements, part of source program A, may require that addresses 01000 and 02000 be changed to 01250 and 02500, respectively.

Useful with long source program



To change these addresses, the operator will enter the following statements at the typewriter



and follow them by source program A. The first time labels CONTRL and NDIGIT are encountered by the processor, they are assigned addresses; the second time they are encountered, no addresses are assigned but an error message is typed. In this case, the operator can ignore the error message.

SEND vs. DEND

In some cases, the programmer may choose to end a source program with a SEND statement rather than a DEND statement. This situation allows the programmer to enter additional statements, either additional declarative statements or corrections to imperative statements. However the last additional statement entered must be a DEND statement.

HEAD — HEADing

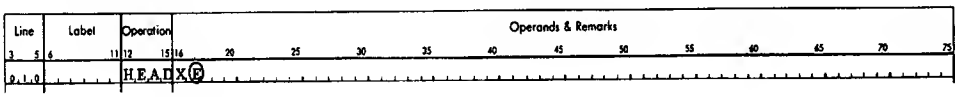
For combining independently written programs

It is frequently convenient, and sometimes necessary, to write a source program piecemeal, and to assemble these pieces into the total program. Parts of the program may be written by different programmers, or by the same programmer at different times with considerable time lapses between.

Suppose, in such a situation, that a program block, say B_1, has been written; that another program block, B_2, is in the course of being written; and that B_1 and B_2 eventually are to be joined to compose a single program. Certain symbols may already have been used in writing block B_1, and certain symbols, varying from the symbols used in B_1, may be used in writing block B_2. To avoid duplication of symbols in each block, the programmer writing block B_2 must be concerned with the symbols used in B_1.

Avoids duplicate symbols

Symbols used in block B_2 can duplicate those in B_1, provided they are less than six characters in length and have been prefaced by a HEAD statement. The programmer can completely ignore the symbols in B_1 by prefacing B_2 with the following control statements:



Adds unique character to label

where the single character X may be any one of the characters A to Z, 0 to 9, or blank.

Generates no instructions or data

The control instruction, HEAD X, generates no instructions or data in the object program. When the processor encounters a HEAD statement, it treats the symbols in the label or operands fields of the following statements, provided the symbols are less than six characters in length, as though they were headed by the character X. The processor continues to do this until it encounters another head.

Thus the symbols used in block B_1 which contain less than six characters cannot possibly conflict with the symbols used in block B_2. Six-character symbols are not

affected, that is, a six-character label, COMMON, following the control instruction HEAD 9 is not treated as 9COMMON, for it would be a seven-character symbol, and only a maximum of six characters can be handled by the symbol table.

All six character symbols unheaded

A symbol is said to be "unheaded" if, and only if, its representation uses exactly six characters. The symbol COMMON, for example, is unheaded. The symbol ALPHA whose length is less than six characters is considered to be headed, whether under a HEAD control instruction or not. If ALPHA is under control of HEAD X, then ALPHA is said to be "headed by X." If ALPHA is not under control of any HEAD instruction, then ALPHA is said to be "headed by blank."

Headed by X
Headed by blank

A symbol, ALPHA, headed by the character X, is not identical with the symbol XALPHA. The heading character is essentially on a different level from the characters which make up the symbol. However, ALPHA headed by a blank should be regarded as identical to the symbol ALPHA used without a heading statement.

Not required for all programs

If a HEAD statement with a nonblank character does not occur in the entire source program, all considerations of heading can be ignored. This is the reason for not introducing the concept of headed symbols earlier.

A HEAD statement with a blank character must be used if the programmer desires to modify the heading process in the example. Note that the statement HEAD and the statement HEAD 0 are quite different. For example, if blocks B_1 and B_2 are to be joined in one program, and B_2 must be nested somewhere in the middle of B_1, as follows:

Nesting

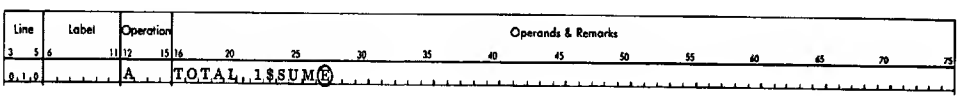
Operation	Operands	
.	.	
.	.	} first part of block B_1
.	.	
HEAD	X	
.	.	
.	.	} block B_2
.	.	
HEAD		
.	.	
.	.	} second part of block B_1
.	.	

the entire program might have been prefaced by a HEAD statement with a blank character operand. As implied previously, however, such a HEAD instruction is superfluous, since the symbols in the first part of block B_1 are automatically headed by blank, being under the control of no HEAD instruction at all.

Cross-referencing headed blocks

Often it is inconvenient to refer to a symbol that is defined in another headed region because of the requirement that the symbol be six characters in length. To facilitate cross referencing between headed blocks, the following convention can be used:

Suppose that a symbol, say SUM, under HEAD 1, has been defined by some instruction. Suppose further that this symbol is to be referred to in an instruction under the control of the instruction HEAD 2. Then the desired reference can be made by writing 1$SUM as it appears in the following instruction



$ signals head character

In general, if the two-characters "C$", where C is any allowable heading character, is placed in front of the headed reference symbol SUM, then the result is SUM headed by C. To specify SUM headed by blank, one simply writes $SUM, with no character preceding the $ character.

Illegal uses

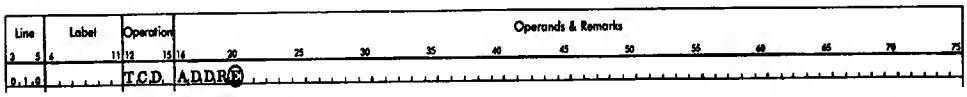
If the processor finds an operand containing a six-character symbol plus a head character, such as 9COMMON, the processor will produce an error message indicating that the symbolic address contains more than six characters (see ERROR MESSAGES, ER 5).

If a label is used in a HEAD statement, it is ignored.

TCD — Transfer Control and loaD

Unconditional branch

The TCD statement may be used to cause the processor to produce an unconditional branch instruction. When this instruction is encountered during the loading of the object (machine language) program, it causes the processor to halt the normal loading process and to branch to the location (ADDR) specified in the operand.



ADDR may be actual or symbolic.

Executes portions of the object program

This statement allows programs which are too large to fit into core storage to be loaded and executed piecemeal, by terminating each piece with a TRA statement. In effect, a TCD instruction can be used in conjunction with a DORG statement to execute portions of the program that have already been loaded into storage and to overlap these with other instructions.

Whenever the processor encounters a TCD statement it causes the arithmetic tables, an unconditional branch instruction, and the loader program, to be punched into the object program tape or cards in that order. Therefore, when the object program is being loaded, the arithmetic tables will be loaded into storage before the branch occurs. Because the arithmetic tables are loaded into a portion of storage previously occupied by the loader program, the loader program will be destroyed. However, it will be restored (again loaded into storage) when a TRA statement is encountered in the object program. That statement will be at the end of the portion of the object program that has been executed.

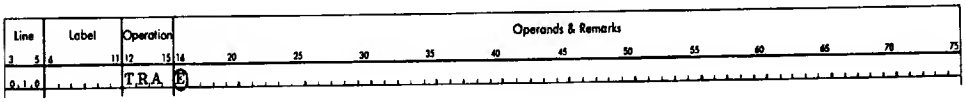
During assembly, the TCD instruction does not affect the location assignment counter or alter the symbol table.

TRA — Transfer to Return Address

Return to loading by TRA

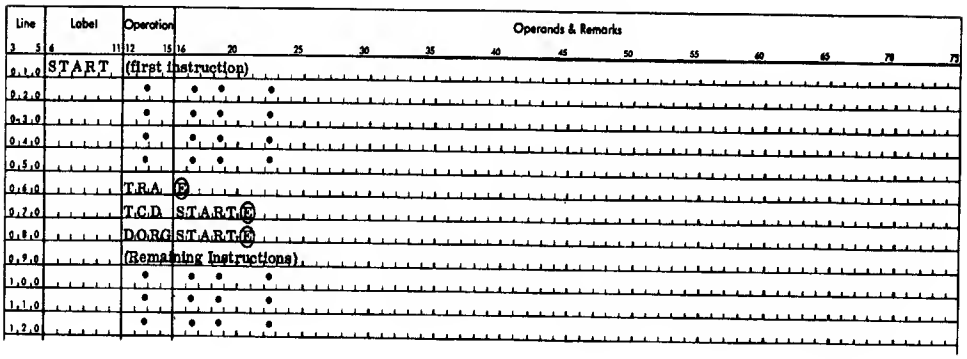
The TRA statement causes the normal loading sequence of an object program to be resumed once it has been broken by a TCD statement. When a TRA statement is encountered by the processor, a read-a-record (card or tape) instruction and an unconditional branch instruction to the loader program are produced in the object program. This processor control operation increments the location assignment counter by 24. The last statement of that part of a source program that is executed, when loading is interrupted by a TCD statement, must be a TRA statement. When the TRA instruction equivalences are encountered in the object program, the program loader is reloaded and the normal loading process continues. The TRA statement, which takes the following form, uses no operands.

No operands



Example

The following example illustrates the use of the TCD and TRA mnemonics.



The TCD statement causes a branch to the location assigned to the symbol START, followed by the execution of instructions from START through the TRA statement. The TRA statement causes a branch to the load program, which resumes loading of the remainder of the object program beginning with the location labeled START.

The use of a macro-instruction preceding a TCD statement is not allowed.

Codes listed in Tables 10 and 11

This section describes the imperative operation codes and "switch or indicator codes" that are used with the 1710 only. Six new operation codes are presented; two of these are classed as branch operations and four as analog-to-digital converter operations. A listing of these operation codes, both mnemonic and actual, is shown in Table 10. Statements using these codes are written in the same manner as 1620/1710 imperative statements. In Table 10, it may be noted that the SAO and SLRN operations require a Q_7 modifier. The list of unique mnemonics in Table 11 may be used in place of SAO and SLRN operations requiring a modifier. Modifiers for unique mnemonics are supplied by the processor.

Table 10. Additional Imperative Operation Codes for the 1710.

TYPE	OPERATION CODE		OPERATION
	MNEMONIC	ACTUAL	
Analog-to-Digital Converter	*SAO	84	Select Address and Operate
	*SLRN	86	Select Read Numerically
	UMK	46	Unmask Interrupts (code zero in Q_{11})
	MK	46	Mask Interrupts (code one in Q_{11})
Branch	BO	47	Branch out of noninterruptible mode, load IR3 with P address (code zero in Q_{11})
	BOLD	47	Branch out of noninterruptible mode, load IR1 with P address (code one in Q_{11})

*Requires coding of modifier in Q_7 or unique mnemonic may be used which automatically generates modifier (see Table 11).

Table 11. Unique Mnemonics To Replace 1710 SAO and SLRN Mnemonics Requiring Modifiers in Q_7.

MNEMONICS		MODIFIER Q_7	OPERATION
EQUIVALENT	UNIQUE		
SAO	SA	1	Select Address
	SACO	2	Select Address and Contact Operate
	SAOS	3	Select Address and Provide Output Signal
SLRN	SLTA	1	Select TAS
	SLAR	2	Select ADC Register
	SLTC	4	Select Real-Time Clock
	SLAD	6	Select ADC and Increment (1711, Model 1, only)
	SLCB	7	Select Contact Block
	SLME	8	Select Manual Entry Switches

Switch or indicator codes, Table 12

Both the BI or BNI operations require as a Q address one of the 14 switch or indicator codes (for 1620/1710) from Table 5 or one of the 35 switch or indicator codes (for 1710 only) from Table 12. These codes indicate the switch or indicator to be interrogated for status by the BI or BNI operations.

Table 12. Additional Switch and Indicator Codes Used as Actual Q Addresses in 1710 BI and BNI Instructions.

Q ADDRESS					SWITCH OR INDICATOR
Q_7	Q_8	Q_9	Q_{10}	Q_{11}	
See note below	0	0	See note below	See note below	Branch Out of Interrupt
	0	8			MAR Check*
	1	8			Operator Entry
	2	1			Terminal Address Selector (TAS) Check*
	2	2			Functional Register Check*
	2	3			Analog Output (AO) Check*
	2	6			Mask Indicator
	2	7			Customer Engineer (CE) Interrupt
	2	8			Analog Output (AO) Setup
	2	9			Terminal Address Selector (TAS) Busy
	4	0			Multiplexer Complete
	4	8			Process Interrupt 1
	4	9			Process Interrupt 2
	5	0			Process Interrupt 3
	5	1			Process Interrupt 4
	7	0			Process Branch Indicator 1
	7	1			Process Branch Indicator 2
	7	2			Process Branch Indicator 3
	7	3			Process Branch Indicator 4
	7	4			Process Branch Indicator 5
	7	5			Process Branch Indicator 6
	7	6			Process Branch Indicator 7
	7	7			Process Branch Indicator 8
	7	8			Process Branch Indicator 9
	7	9			Process Branch Indicator 10
	8	0			Process Branch Indicator 11
	8	1			Process Branch Indicator 12
	8	2			Process Branch Indicator 13
	8	3			Process Branch Indicator 14
	8	4			Process Branch Indicator 15
	8	5			Process Branch Indicator 16
	8	6			Process Branch Indicator 17
	8	7			Process Branch Indicator 18
	8	8			Process Branch Indicator 19
	8	9			Process Branch Indicator 20

*The Any Data Check indicator (19) also is on when this indicator is on.

NOTE: Position Q_7 may contain any digit value or blank, Q_{10} may contain any digit value, and Q_{11} any digit value with the exception of zero or one.

Program or Routine

A program or routine is a set of coded instructions that are arranged in a logical sequence; it is used to direct the 1620, or any IBM data processing system, to perform a desired operation or series of operations. Generally, programs contain one or more short sequences of instructions that are parts or subsets of the entire program and that are used to solve a particular part of a problem. These parts of the program or routine are called *subroutines.*

Common to many programs

Subroutine

Usually, a subroutine performs a specific function, is common to a number of programs, and may be executed several times during the course of the program of which it is a part (main program). For example: a subroutine that extracts the square root of a number may be required several times during the execution of a pipe stress analysis program. The same subroutine may be used to extract a square root in a bridge and truss design program.

An efficient programming procedure is obviously one in which all necessary subroutines are coded only once, are retained on file, and are incorporated into a program whenever the operation performed by the subroutine is required. IBM Applied Programming has developed for the 1620/1710 Symbolic Programming System a group of subroutines that are more frequently required because of their general applicability. Seventeen subroutines are available; they fall into three general categories: arithmetic, data transmission, and functional.

17 Library subroutines

Arithmetic subroutines
- Floating Point Add
- Floating Point Subtract
- Floating Point Multiply
- Floating Point Divide
- Fixed Point Divide

Data transmission subroutines
- Floating Shift Right
- Floating Shift Left
- Transmit Floating
- Branch and Transmit Floating

Functional subroutines (those that evaluate)
- Floating Point Square Root
- Floating Point Sine
- Floating Point Cosine
- Floating Point Arctangent
- Floating Point Exponential (natural)
- Floating Point Exponential (base 10, common)
- Floating Point Logarithm (natural)
- Floating Point Logarithm (base 10, common)

The methods used by the functional floating point subroutines to evaluate the functions of arguments are shown in Table 13.

Variable length
Fixed length

The combined subroutines are written in machine language and are provided in card or paper tape form for floating point numbers with either a fixed length or variable length mantissa. The term "variable length" or "fixed length," as applied to subroutines in this manual, refers to the number of digits (L) in the mantissa, not to the length of the subroutine itself.

Table 13. Subroutine Method for Evaluating Arguments.

SUBROUTINE	METHOD	
	FIXED LENGTH	VARIABLE LENGTH
Square Root	Odd integer	Odd integer
Sine and Cosine	Based on Hastings' approximation*	Series approximation
Arctangent	Truncated series	Series approximation for arctangent
Exponential (natural and base 10)	Hastings' approximation for 10^B 10^B is converted to e^B	Series approximation of 10^B and convert to e^B
Logarithm (natural and base 10)	Truncated series for ln B ln B is converted to $\log_{10}$ B	Series approximation of ln B and convert to log B

* Hastings, Cecil, Jr., *Approximations for Digital Computers*, Princeton University Press, Princeton, New Jersey, The Rand Corporation, 1955.

Five sets of subroutines

The five types of subroutine card decks or paper tapes are:

1. *Fixed* length subroutines for machines *not* equipped with the automatic divide feature.
2. *Fixed* length subroutines for machines equipped with the automatic divide feature.
3. *Variable* length subroutines for machines *not* equipped with the automatic divide feature.
4. *Variable* length subroutines for machines equipped with the automatic divide feature.
5. *Variable* length subroutines for machines equipped with automatic floating point feature (for which the automatic divide feature is a prerequisite).

Although type 2 and type 4 subroutines are designed to work with the automatic divide feature, the "fixed point divide" subroutine is included as part of the subroutine package. Type-5 variable length subroutines that are designed to work with the automatic floating point feature include a complete set of subroutines as part of the package.

PICK common to all subroutines

A PICK subroutine is included in the object program with any of the 17 subroutines previously mentioned. This subroutine performs the function of getting the data specified for a subroutine and storing the result produced by that subroutine.

Function of processor

The processor selects the subroutines used by the source program that are to be included in the object deck or tape. When the object deck or tape is loaded, the subroutines are loaded to the first even-numbered address following the object program. Although this address is assigned by the processor, care must be exercised by the programmer to provide a storage area, between the position assigned by the processor and position 19999 (standard-capacity machine), that is large enough to accommodate the subroutines called for. To find the amount of storage required for the subroutines, the programmer may total the storage requirements of the subroutines used.

The fixed point divide subroutine (DIV) is used by some floating point functional subroutines. For this reason it will automatically be incorporated into a program which uses these subroutines when the machine used to run the program is not equipped with automatic divide (special feature).

Adding subroutines

In addition to the Library subroutines, the user may include up to twelve subroutines of his own. The method used to incorporate these routines into a program is explained under ADDING SUBROUTINES.

Linkage and Macro-instructions

Programmer writes macros

Processor generates linkage

All linkages for the 1620 subroutines are generated automatically through the use of certain macro-instructions. The programmer places the macro-instruction that is related to a particular subroutine in the source program at the point at which the subroutine is desired. This causes the SPS processor, during assembly, to generate linkage to the desired subroutine. In addition, the processor arranges for the subroutine to be added to the object program.

The data and addresses required by the subroutine and supplied in the macro-instruction are incorporated into the linkage instructions where they are made available for use. In this way the subroutine obtains the information it requires to perform its given task and also to compute a return address to the main program. Control is returned to the main program at the completion of the subroutine by transferring to the return address. The macro-instruction statement related to each subroutine is as follows:

Arithmetic Subroutines Macro-instruction

Line	Label	Operation	Operands & Remarks
010		FA	A, BⒺ (Floating Add)
020		FS	A, BⒺ (Floating Subtract)
030		FM	A, BⒺ (Floating Multiply)
040		FD	A, BⒺ (Floating Divide)
050		DIV	A, B, A1, B1Ⓔ (Divide)

Data Transmission Subroutines Macro-instruction

Line	Label	Operation	Operands & Remarks
010		FSRS	A, BⒺ (Floating Shift Right)
020		FSLS	A, BⒺ (Floating Shift Left)
030		TFLS	A, BⒺ (Transmit Floating)
040		BTFS	A, BⒺ (Branch and Transmit Floating)

Functional Subroutines Macro-instruction

Line	Label	Operation	Operands & Remarks
010		FSQR	A, BⒺ (Floating Square Root)
020		FSIN	A, BⒺ (Floating Sine)
030		FCOS	A, BⒺ (Floating Cosine)
040		FATN	A, BⒺ (Floating Arctangent)
050		FEX	A, BⒺ (Floating Exponential, Natural)
060		FEXT	A, BⒺ (Floating Exponential, Base 10)
070		FLN	A, BⒺ (Floating Logarithm, Natural)
080		FLOG	A, BⒺ (Floating Logarithm, Base 10)

Arithmetic subroutine macros

Data transmission macros

Functional subroutine macros

In the arithmetic statements, the B operands represent the addresses of quantities to be added, subtracted, etc., to quantities at addresses specified by the A operands. For the fixed point divide routine, two additional operands, A1 and B1, are required. These operands, as well as the A and B operands, are explained in greater detail under each macro-instruction as it is described. In the data transmission statements, the B operand generally represents the address of the field to be transmitted, whereas the A operand represents the address to which the field is to be transmitted. The function of the A and B operands differs slightly for functional subroutine macro-instructions. In this case, the B operand represents the

Use of indirect addressing with macros

address of the argument to be evaluated and the A operand represents the address where the result is placed in storage.

Indirect addressing (special feature) *can* be used with the operands of all macro-instructions on machines with or without the automatic floating point feature. To indicate an indirect address, an operand should be preceded by a minus sign. An indirect address in the form $\bar{x}xxx\bar{x}$ is generated by the processor. A flag is automatically placed in the leftmost and rightmost positions of the address. A flag in any other position of an indirect address is not permitted. If indirect addressing is attempted on a machine that does not have the indirect addressing feature, the flagged operand is *not* interpreted as an indirect address.

Two linkage instructions for each macro

For each macro-instruction statement in a source program, two machine language linkage instructions, and a 5-digit address for each operand, are generated by the processor in the object program. These linkage instructions replace the macro-instruction, which never appears in the object program. A label written with a macro-instruction references the leftmost position of the first linkage instruction generated. If the programmer wishes to use this label in address adjustment, he must remember that the instruction located following a macro-instruction is not LABEL + 12.

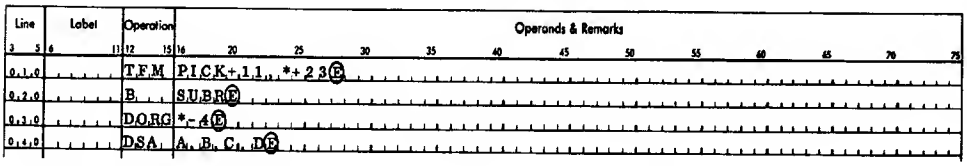
Rules for coding macros

When using a macro-instruction, the programmer must code the exact number of operands required for that macro-instruction. Every macro-instruction must have at least two operands. Remarks and flag operands are not permitted in macro-instructions. Omitted operands require the insertion of commas as in imperative statements.

All operands in macro-instructions may be symbolic or actual; all are subject to address adjustment. If an * is used as an operand, its address is that of the leftmost postion of the first linkage instruction.

Linkage

The linkage instructions generated for a macro-instruction by the processor are *equivalent* to the following series of symbolic instructions:



where PICK is the address of the first instruction in the pick subroutine that is shared by all subroutines; SUBR is the secondary linkage for the desired subroutine (that subroutine specified by the macro-instruction); and A, B, C, D . . . are the series of 5-digit addresses or constants that are equivalent to the operands specified by the macro-instruction. This linkage allows the macro-instruction to contain any number of operands, an ability that is significant for "adding subroutines."

Secondary linkage

During execution of the object program, the secondary linkage instructions set up the address of the first instruction in the desired subroutine as a part of one of the instructions in the pick subroutine. Secondary linkage instructions are generated by the processor for each subroutine used by the source program. They are equivalent to the following symbolic instructions:

For *arithmetic* subroutines (not including DIV)

```
SUBR  TFM   PICK + 402, ADDR (E)
      B     PICK (E)
```

For *data transmission* subroutines (not including FSRS, FSLS)

```
SUBR  TFM   PICK + 402, ADDR (E)
      B     PICK + 104 (E)
```

For *functional* subroutines

```
SUBR  TFM   PICK + 402, ADDR (E)
      B     PICK + 104 (E)
```

For DIV (arithmetic), FSRS, and FSLS (data transmission) subroutines

```
SUBR  TF    ADDR + 11, PICK + 11 (E)
      B     ADDR (E)
```

PICK is the address of the first instruction of the pick subroutine, and ADDR is the actual address where the first instruction of the desired subroutine is located. In the secondary linkage, PICK + 402 is the address equivalent for the variable length subroutines only. That address should be replaced by PICK + 414 for fixed length subroutines. For variable length subroutines using the floating point feature, the secondary linkage instructions generated are equivalent to the following symbolic instructions:

For *arithmetic* subroutines

```
SUBR  TF    ADDR + 11, PICK + 11 (E)
      B     ADDR (E)
```

For *data transmission* subroutines (not including FSRS, FSLS)

```
SUBR  TF    PICK + 174, ADDR (E)
      B     PICK + 24 (E)
```

For *functional* subroutines

```
SUBR  TF    PICK + 174, ADDR (E)
      B     PICK + 24 (E)
```

Subroutines contained in deck or tape

The subroutine card deck or paper tape will contain the subroutines in the order shown. All except the first three are floating point subroutines.

1. Subroutine Processor
2. Pick
3. Divide (fixed point)
4. Subtract – Add
5. Multiply
6. Divide
7. Square Root
8. Cosine – Sine
9. Arctangent
10. Exponential (natural and base 10)
11. Logarithm (natural and base 10)
12. Shift Right
13. Shift Left
14. Transmit Floating
15. Branch and Transmit Floating

Paired subroutines

Conserve storage

Many subroutines have been paired (i.e., add and subtract, sine and cosine, natural and base 10 exponential, natural and base 10 logarithm), into single subroutines to conserve storage by sharing those program steps common to both. The individual subroutines within each pair are distinguished from each other solely by the point at which they are entered. The correct entry point is obtained through use of the macro-instruction pertaining to the particular subroutine desired.

Divide subroutine used by certain subroutines

Because the arctangent subroutine and both logarithm subroutines (natural and base 10) require the fixed point divide routine, the macro-instructions FATN, FLN, and FLOG cause the fixed point divide subroutine to be added to the object program output (provided the machine is not equipped with automatic divide). For the fixed length mantissa subroutines, any of the previously listed subroutines 3-8, 10, 12-15, may be called for and added to the object output without calling any other subroutine; i.e., floating add-subtract may be called without calling floating multiply. For the variable length mantissa subroutines, any of the four arithmetic macro-instructions (FA, FS, FM, FD) will cause all three arithmetic subroutines (4. floating add-subtract, 5. floating multiply, and 6. floating divide) to be called for and added to the object program output. However, the other subroutines may be called for independently of each other.

In the object program output, each subroutine except the pick subroutine is preceded by its secondary linkage. However, when the object program is loaded, the secondary linkages for all of the subroutines are stored in storage positions that immediately follow the object program, starting with the first even-numbered address. PICK and other subroutines are loaded into sequentially higher addresses (in the order previously listed).

Subroutine processor

The subroutine processor is loaded into an area of storage which is occupied by the SPS processor, thereby destroying a portion of the SPS processor. To restore the SPS processor to its original status, the part destroyed must be reloaded after selection of the subroutines is completed. Restoration of the SPS processor is accomplished automatically with the data which follows the last subroutine (15. Branch and Transmit Floating) of the subroutine deck or tape. This data includes the loader, arithmetic tables, and that part of the SPS processor previously destroyed. The "subroutine processor" is never a part of the object program output or final object program.

Not part of object program

Sequence Numbering of Subroutines

Subroutine sequence maintained by number in columns 77-80

The subroutine deck contains a sequence number in columns 77-80. This number allows the programmer to restore the correct sequence should a deck be dropped or a card inadvertently misplaced. To operate correctly, subroutines should be in

Table 14. Codes Used in Column 77 of Subroutine Deck To Identify and Sequence Subroutines.

SUBROUTINE	NUMBER	CODE IN COLUMN 77	
		FIXED LENGTH	VARIABLE LENGTH
Subroutine Processor	None	blank	blank
PICK	00	0	0
DIV	01	1	1
FS	02	2	2
FA	03		
FM	04	4	
FD	05	5	
FSQR	06	6	6
FCOS	07	7	7
FSIN	08		
FATN	09	9	9
FEXT	10	$\bar{0}$	$\bar{0}$
FEX	11		
FLOG	12	$\bar{2}$	$\bar{2}$
FLN	13		
FSRS	14	$\bar{4}$	$\bar{4}$
FSLS	15	$\bar{5}$	$\bar{5}$
TFLS	16	$\bar{6}$	$\bar{6}$
BTFS	17	$\bar{7}$	$\bar{7}$

sequence. Subroutines not wanted by a user can be removed from the subroutine deck. The macro-instruction associated with that subroutine may not be used, if the subroutine has been removed.

Fixed length and variable length subroutine codes, Table 14

Each card in the subroutine deck contains a code in column 77. These codes for both fixed length and variable length subroutines are shown in Table 14. In addition, each subroutine is numbered 000, 001, etc., in columns 78-80. A flag in column 78 indicates variable length subroutines, whereas omission of this flag indicates fixed length subroutines. A flag in column 79 indicates that the subroutine is designed to work with automatic divide; no flag in this column indicates that it is designed to work without the automatic divide feature. A flag in column 80 indicates it is designed to work with the automatic floating point feature.

Floating Point Arithmetic

Scientific and engineering computations frequently involve lengthy and complex calculations necessitating the manipulation of numbers that may vary widely in magnitude. To obtain a meaningful answer, problems of this type usually require retention of as many significant digits as possible during calculation, and correct positioning of the decimal point at all times. When the computer is used for such problems, several factors must be considered, of which the most important is the location of the decimal point.

To locate decimal point

In general, a computer does not recognize the presence of a decimal point in any quantity during calculation. A product of 41454 results whether the factors are 9.37 x 44.2; 93.7 x .442; or 937 x 4.42; etc. The programmer must be cognizant of the location of the decimal point during and after the calculation and arrange the program accordingly. In adding, the decimal points of all numbers must be lined up to obtain the correct sum. The programmer facilitates this arrangement by shifting the quantities as they are added. In the manipulation of numbers that vary greatly in magnitude, it is conceivable that the resulting quantity could exceed allowable working limits.

To handle large numbers

Processing numbers expressed in ordinary form, e.g., 427.93456, 0.0009762, 5382, −623.147, 3.1415927, etc., can be accomplished on a computer only with extensive analysis to determine the size and range of intermediate and final results. The percentage of time required for this analysis and subsequent number scaling is frequently much larger than the percentage of time required to perform the actual calculation. Moreover, number scaling requires complete and accurate information regarding the bounds on the magnitude of all numbers that come into the computation (input, intermediate, output). Since prediction of the size of all numbers in a given calculation is not always possible, analysis and number scaling are sometimes impractical.

Eliminates scaling

To alleviate this programming problem, a system must be employed which provides information regarding the magnitude of all numbers in the calculation along with the quantities in the calculation. Thus, if all numbers are represented in some standard, predetermined format that instructs the computer in an orderly and simple fashion as to the location of the decimal point, and if this representation is acceptable to the routine that performs the calculation, then quantities that range from minute fractions having many decimal places to large whole numbers having many integer places can be handled. The arithmetic system most commonly used, in which all numbers are expressed in a format that has these characteristics, is called "floating point arithmetic."

Standard representation for all quantities

The notation used in floating point arithmetic is basically an adaptation of the scientific notation that is widely used today. In scientific work very large or

Scientific notation

very small numbers are expressed as a number, between one and ten, times a power of ten. Thus,

427.93456 is written as 4.2793456×10^{2},
and 0.0009762 is written as 9.762×10^{-4}

In the 1620 floating point arithmetic system, the range of the fractional part of the number is modified to extend between .10000000 and .99999999, that is, the decimal point of all numbers is placed to the left of the high-order (leftmost) nonzero digit. Hence, all quantities may be thought of as a decimal fraction times a power of ten. For example,

427.93456 becomes $.42793456 \times 10^{3}$
and 0.0009762 becomes $.97620000 \times 10^{-3}$

Mantissa
Exponent

where the fraction is called the *mantissa*, and the power of ten, indicating the number of places the decimal point was shifted, is called the *exponent*. The use of floating point numbers during processing, besides offering advantages inherent in scientific notation, eliminates the need for analyzing operations in order to determine the positioning of the decimal point in intermediate and final results, since the decimal point is always immediately to the left of the high-order, nonzero digit in the mantissa.

Format

In 1620/1710 floating point operations, a floating point number is a field consisting of a variable length or fixed length mantissa and a 2-digit exponent. The exponent is in the two low-order positions of the field, and the mantissa is in the remaining high-order positions, as shown:

$\overline{M}$ . . . . . M$\overline{E}$E

Limits

For the subroutines, the variable length mantissa may have a minimum of two digits and a maximum of 45 digits. Two operand fields that are added together must have mantissas of the same length. A flag over the high-order digit marks the extremity of the field. A fixed length mantissa must have eight digits.

Range of exponent

The exponent is established on the premise that the mantissa is less than 1.0 and equal to or greater than 0.1. It always consists of two digits ranging between −99 and +99. A flag over the high-order (ten) digit defines the exponent.

The high-order digit of the mantissa and the high-order digit of the exponent *must* contain flag bits to operate properly with floating point subroutines.

Sign control

The mantissa and the exponent, if negative, must have an algebraic sign, represented by a flag, over the units position of the respective fields; if they are positive, they are not flagged. A floating point number with a negative mantissa and a negative exponent is represented as follows:

$\overline{M}$ . . . . . $\overline{M}\overline{E}\overline{E}$

Sign control of the results of all computations is maintained according to the standard rules of arithmetic operations.

Normalized
Unnormalized

In all floating point numbers the decimal point is assumed to be at the left of the high-order digit, which must be a nonzero digit. Such a number is referred to as *normalized*. When a number has one or more high-order zeros, it is considered to be *unnormalized*. An unnormalized number resulting from a floating point subroutine computation is normalized automatically, but unnormalized terms are not recognized as such when entered as data. Therefore, it is necessary for all data to be entered in normalized form. Although unnormalized numbers will be processed, correct results cannot be assured. For example, the number $\overline{0}682349\overline{4}\overline{0}\overline{5}$ should be entered as $\overline{6}823494\overline{0}\overline{0}4$, assuming the fixed point number is 6823.494, and an 8-digit mantissa is required.

Data must be normalized

Conversion

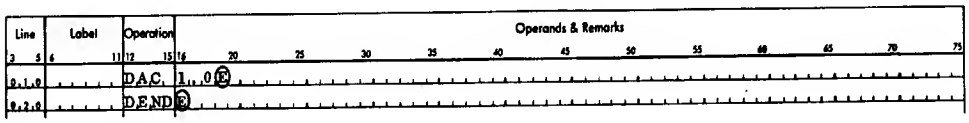
The following examples demonstrate the conversion of numbers in ordinary form to 1620 floating point notation for an 8-digit mantissa.

NUMBER	NORMALIZED	1620 FLOATING POINT
123.45678	$.12345678 \times 10^{3}$	$\bar{1}234567\bar{8}\bar{0}3$
.00765438	$.76543800 \times 10^{-2}$	$\bar{7}654380\bar{0}\bar{2}$
—.12348693	$-.12348693 \times 10^{0}$	$\bar{1}234869\bar{3}\bar{0}0$
—.00000070	$-.70000000 \times 10^{-6}$	$\bar{7}000000\bar{0}\bar{6}$
.00000000	$.00000000 \times 10^{-99}$	$\bar{0}000000\bar{9}\bar{9}$

NOTE: A zero mantissa is associated with a $\bar{9}\bar{9}$ exponent.

N digit

The result of a floating point operation is normalized automatically. For example, the result .00123456 when normalized becomes $\bar{1}$23456NN$\bar{0}\bar{2}$, where N is an inserted digit and $\bar{0}\bar{2}$ is the exponent. The value of the N digit (0 through 9) is determined by the programmer, who in most cases will choose to use zero. The storage location of the N digit is the first odd-numbered location following the last secondary linkage of the assembled program. *The programmer must always store the N digit.* He may do this by using the following statements, where the constant (N digit) is zero.



Effects of normalizing

In normalizing, certain low-order digits in a mantissa may lose significance. To recognize these digits, the floating point arithmetic can be performed twice, using a different N digit for each run, e.g. zero for the first run and nine for the second run. The significance of these digits can be readily distinguished by comparing the two results. For example, if the programmer compares the following:

	Mantissa	Exponent
Result, 1st run	.12345000	04
Result, 2nd run	.12345099	04

he will see that the two low-order positions of the mantissa have lost significance because they are significantly different.

When intermediate floating point results enter into additional floating point calculations, inserted digits may become a part of the result of the additional calculation.

Truncation error

In the case of lengthy computations using floating point results, precision gradually decreases because of truncation. The magnitude of the truncation error depends on the individual computation process and cannot be predicted without a knowledge of the process in question. However, the truncation error in such cases is usually no greater than the degree of error present in a rounded amount. Results in floating point subroutines are not rounded. The maximum truncation error for a fixed length mantissa will not exceed 10^{-8} or for a variable length mantissa, 10^{-L}, except under certain conditions described in the explanation of floating point functional subroutines.

Maximum truncation error

Exponent Overflow and Underflow

Overflow
Underflow

In the 1620/1710 floating point subroutines, numbers with a magnitude equal to or greater than 10^{99} create a condition called *exponent overflow;* those with a magnitude of less than 10^{-99} create a condition called *exponent underflow.* If either of

these conditions is generated as a result of an arithmetic operation, the programmer has two options.

Overflow

1. To halt the program or
2. To cause $\bar{9}$..... $9\bar{9}9$ to be placed in the result field and to continue executing the subroutine.

Underflow

1. To halt the program or
2. To cause $\bar{0}$..... $0\bar{9}\bar{9}$ to be placed in the result field and to continue executing the subroutine.

Options

The options function independently of each other. Therefore it is possible to halt on an overflow and place zeros in the result field on an underflow, or to halt on an underflow and place nines in the result field on an overflow.

Determined by position 401

The detection of an overflow or underflow condition causes the subroutine being executed to examine core storage position 00401 to determine the course of action. Options available to the programmer must be represented in position 401 by one of the following characters:

		UNDERFLOW	
		Halt	Store Zeros in Result Field
OVERFLOW	Halt	$\bar{0}$	0
	Store All Nines in Result Field	$\bar{1}$	1

To store the code determining the option in 401, an unlabeled Define Constant (DC) statement may be used, as shown in the following example:

```
Line  Label  Operation  Operands & Remarks
2 5   6   11 12  15    16  20  25  30  35  40  45  50  55  60  65  70  75
0,1,0        DC         2,,-0,,4,0,1,,HALT ON OVERFLOW OR UNDERFLOW®
```

Positive codes (0 or 1) need not be preceded by a plus sign.

Affects six subroutines

Overflow and/or underflow conditions can only arise in six of the floating point subroutines presented in this manual, namely, the four arithmetic subroutines and the two exponential functional subroutines.

When subroutine halts

If the subroutine halts on an overflow or underflow condition, the operator can continue processing by depressing the start key on the console. In the case of an overflow, execution of the subroutine begins after $\bar{9}$ ..... $9\bar{9}9$ is placed in the result field; in the case of an underflow, after $\bar{0}$..... $0\bar{9}\bar{9}$ is placed in the result field.

Subroutine Error Messages

Format of error messages

In 1620 subroutines the presence of special conditions causes an error message. This message is typed out in the following form:

XXXXX00XX
R S

where R is a return address to the main program
and S is a code that identifies the special condition.

Error codes

With the exception of exponent overflow or underflow, where the course of action depends upon the digit at location 401, a subroutine always halts immediately after typing the error message. The error message code indicates the reason for the halt. The operator may insert a branch instruction at storage location 00000. The branch instruction will contain the return address to the main program. In cases such as floating square root, execution of the subroutine may be made to continue, after a halt, by depressing the start key.

Listed in Table 15

Table 15 lists the error codes for special conditions and the appropriate action to be taken.

Table 15. Error Codes

ERROR CODE	DESCRIPTION OF ERROR	OPERATOR'S ACTION WHEN SUBROUTINE HALTS
01	FA or FS, Exponent Overflow	May continue execution of subroutine by depressing start key
02	FA or FS, Exponent Underflow	Same as code 01
03	FM, Exponent Overflow	Same as code 01
04	FM, Exponent Underflow	Same as code 01
05	FD, Exponent Overflow	Same as code 01
06	FD, Exponent Underflow	Same as code 01
07	FD, Attempt to divide using a number with a zero mantissa divisor	May not continue execution of subroutine but may branch back to main program using return address
08	FSQR, Attempt to find the square root of a negative number	Pressing the start key causes the subroutine to extract the square root of the absolute value of the argument
09	FSIN or FCOS, Input argument has an exponent value greater than $\bar{0}8$ (fixed length mantissa) or L (variable-length mantissa)	Same as code 07
10	FSIN or FCOS, For a fixed-length mantissa, the input argument has an exponent (X) such that $\bar{0}3 \leq X \leq \bar{0}8$. For a variable-length mantissa, the input argument has an exponent (X) such that $\bar{0}3 \leq X \leq L$ where L = length of a mantissa.	Same as code 01
11	FEX or FEXT, Exponent Overflow	Same as code 01
12	FEX or FEXT, Exponent Underflow	Same as code 01
13	FLN or FLOG, Input argument has a zero mantissa	Same as code 07
14	FLN or FLOG, Input argument is negative	Pressing the start key causes the subroutine to continue execution, using the absolute value of the argument

Storage requirements

In this section are described the various subroutines, their associated macro-instructions, core storage requirements, and average execution time. By totaling the storage required for each subroutine in a specific program, the programmer can ascertain whether there is sufficient storage between the last position of storage that is used by the object program and position 19999 (standard-capacity machine) to store the subroutines. As stated earlier, the four subroutines that use fixed point divide (either automatic divide or the divide subroutine) are: floating divide, floating arctangent, floating exponential, and floating logarithm. Fixed length mantissa subroutines that use automatic divide in the computation require less storage than fixed length mantissa subroutines that use the fixed point divide subroutine.

Listed in Table 16

Table 16 shows the storage requirement for each fixed length and variable length subroutine.

Table 16. Summary of Storage Requirements of Subroutines.

SUBROUTINE	NUMBER OF STORAGE POSITIONS REQUIRED				
	FIXED LENGTH		VARIABLE LENGTH		
	WITH AUTOMATIC DIVIDE	WITHOUT AUTOMATIC DIVIDE	WITH AUTOMATIC DIVIDE	WITHOUT AUTOMATIC DIVIDE	WITH AUTOMATIC FLOATING POINT
PICK	872	872	1136	1136	896
DIV	187	1047	199	1035	199
FA	543	543	1163	1207	603
FS					
FM	239	239			
FD	335	523			
FSQR	579	579	659	659	659
FCOS	843	843	1054	1098	1054
FSIN					
FATN	989	1077	1379	1487	1379
FEXT	740	784	1118	1258	1118
FEX					
FLOG	842	886	1145	1209	1145
FLN					
FSRS	279	279	279	279	96
FSLS	372	372	372	372	96
TFLS	31	31	31	31	31
BTFS	79	79	79	79	43

High/Positive Equal/Zero indicators

During the execution of *arithmetic* subroutines, the overflow, high/positive, and equal/zero indicators are used. The overflow indicator is always reset at the beginning of each arithmetic subroutine. If it is desired to determine its status prior to the execution of an arithmetic subroutine, the indicator must be tested and its condition stored before the linkage instructions are executed. The high/positive and equal/zero indicators are set according to the mantissa of the result. Whenever a zero mantissa results ($\bar{0}$.....0$\bar{9}\bar{9}$), the equal/zero indicator is turned on.

At the conclusion of a functional subroutine, the status of the high/positive, equal/zero, and overflow indicators does not necessarily reflect the result of the operation, because the indicators are disturbed during the execution of a functional subroutine. Therefore, their status at the conclusion of a functional subroutine should not be assumed to be the same as it was prior to the execution of the subroutine.

Pick

Pick

This subroutine is common to all fixed length and variable length mantissa subroutines. The pick subroutine, during execution of the object program,

Functions

1. Sets up A and B operands (more, if designated) to be operated upon, calculates the return address to the mainline program, and branches to the subroutine.
2. Stores the calculated result in the proper storage area and branches back to the mainline program.
3. Initiates typing of error messages and branches to the subroutine if the error condition allows processing of the subroutine to be resumed.
4. Provides constants and working storage for the other subroutines.

Execution time

The average execution time for the pick subroutine can be determined by the formula:

$$\text{Average time (in } \mu\text{s)} = 100L + 8320$$

where L equals the length of the mantissa and the numbers are expressed in microseconds. Therefore, an 8-digit mantissa (same as fixed length mantissa) requires 9120 μs.

$$\begin{array}{rr} 100 \times 8 = & 800 \\ & 8320 \\ & \overline{9120}\ (\mu s) \end{array}$$

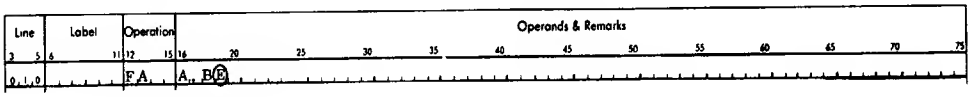
or approximately nine milliseconds (ms). If indirect addressing is used, the average time is increased according to the number of levels of indirect addressing used.

NOTE: For the variable length subroutines used with the automatic floating point feature,

$$\text{Average time (in } \mu\text{s)} = 100L + 4500$$

Floating Add

Macro-instruction



FA macro

The A and B addresses refer to the units position of the exponent of the fields:

$$\bar{M}MMMMMMM\bar{E}E$$
$$\uparrow$$
address of field

where Es represent digits of the exponent and Ms represent digits of the mantissa.

Operation

Field B is added to field A. The floating point sum replaces field A; field B remains unchanged.

Average Execution Time

Fixed length

Average time = 9 ms

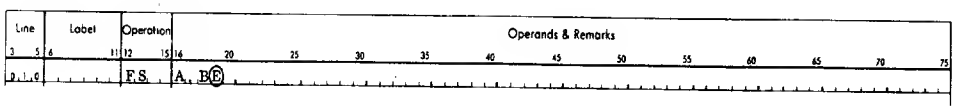
Variable length

Average time (in μs) = $5L^2 + 482L + 6854$

where L = length of mantissa

Floating Subtract

Macro-instruction



FS macro

The A and B addresses refer to the units position of the exponent of the fields.

Operation

Field B is subtracted from field A. The floating point difference replaces field A; field B remains unchanged.

Average Execution Time

Fixed length

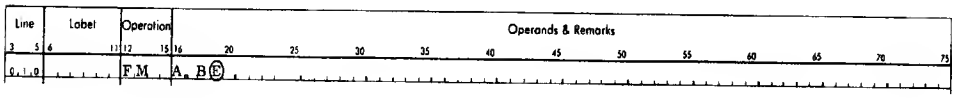
Average time = 10.5 ms

Variable length

Average time (in μs) = $5L^2 + 482L + 7474$

Floating Multiply

Macro-instruction



FM macro

The A and B addresses refer to the units position of the exponents of the fields.

Operation

Field A is multiplied by field B. The floating point product replaces field A; field B remains unchanged.

Average Execution Time

Fixed length

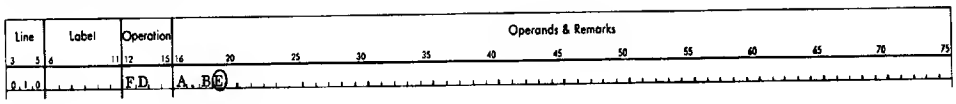
Average time = 18 ms

Variable length

Average time (in μs) = $168L^2 + 240L + 7400$

Floating Divide

Macro-instruction



FD macro

Operation

Field A is divided by field B. The floating point quotient replaces field A; field B remains unchanged.

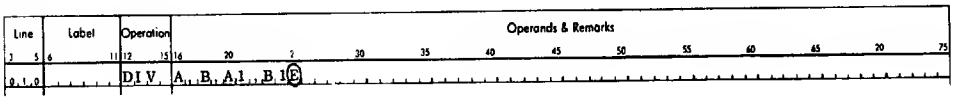
Average Execution Time

Fixed length
- With automatic divide
  - Average time = 55 ms
- Without automatic divide
  - Average time = 70 ms

Variable length
- With automatic divide
  - Average time (in μs) = $520L^2 + 1500L + 7890$
- Without automatic divide
  - Average time (in μs) = $1.9[520L^2 + 1500L + 7890]$

Fixed Point Divide

Macro-instruction



DIV macro

In addition to the A and B operands, representing the addresses of the dividend and divisor, the divide macro-instruction requires two additional operands; one specifies the number of zeros to be inserted to the right of the dividend (A1 operand) and the other, the shift factor needed by the subroutine (B1 operand). Specifically,

Four operands required

A operand is core storage address of dividend.
B operand is core storage address of divisor.
A1 operand is 00099 minus the number of zeros desired to the right of the units position of the dividend.
B1 operand is 00100 minus the length of the quotient. The quotient must be at least two digits in length.

NOTE: The quotient address after the division is executed will be equal to 00099 minus the length of the divisor.

Product area 00080-00099 cleared automatically

Prior to the divide operation, the divide subroutine always resets to zeros (clears) the positions 00080 through 00099, the product area where the 20-digit quotient and remainder are developed. For the variable length mantissa subroutines, where L (length of mantissa) is greater than 10, the number of positions which are reset to zeros is equal to 99 —2L. When the quotient plus the remainder exceeds the number of positions cleared to zeros, positions lower than the last posi-

Additional positions cleared by programming

tion cleared must be reset to zeros by *programming*. One additional position should also be cleared to allow for a possible overdraw. For example, if 25 positions are required for the quotient and remainder in a fixed length mantissa subroutine, 00074-00079 will have to be reset to zeros before the divide macro-instruction is given.

Macro may be used with any subroutine package

The fixed point divide macro-instruction may be used with any of the subroutine packages. Whenever it is used, the fixed point divide subroutine will be incorporated into the user's program. For the subroutine packages that are designed to work with automatic divide, the fixed point divide subroutine uses automatic divide in performing its operation. For the subroutine packages that are designed to work without the automatic divide feature, the fixed point divide subroutine performs its operation as instructed by the routine without the aid of the automatic divide feature. Coding of the macro-instruction is the same for all of the subroutine packages.

Operation

Positioning of dividend

The product area (00080-00099) is automatically reset to zeros. The dividend (A address) is transmitted to the product area (A1 address), beginning at the low-order dividend digit and terminating at the flag bit marking the high-order position of the dividend field. The A1 address is 00099 minus the number of zero positions desired to the right of the dividend.

High-order digit flagged

The algebraic sign of the dividend is automatically placed in location 00099, regardless of where the rightmost dividend digit is placed by the A1 address. A flag bit automatically marks the high-order digit of the dividend.

Positioning of divisor for subtraction

The divisor (B address) is successively subtracted from the dividend. The B1 address of the divide macro-instruction positions the divisor for the first subtraction from the high-order position(s) of the dividend, as in manual division. The B1 address is determined by subtracting the number of digits in the quotient from 100. For the subroutines using program divide, the value of B1 must be between 0 and 99. For subroutines using automatic divide, the value of B1 is not restricted.

Quotient and remainder in product area

The quotient and remainder replace the dividend in the product area. The address of the quotient is 00099 minus the length of the divisor. The algebraic sign of the quotient (determined by the signs of the dividend and divisor) is automatically placed in the low-order position of the quotient. The address of the remainder is 00099 and a flag bit is automatically placed in that position. The remainder has the sign of the dividend and the same number of digits as the divisor.

Sign of result shown by indicators

The high/positive indicator is on if the quotient is positive and not zero; the equal/zero indicator is on if the quotient is zero. Neither indicator is on if the quotient is negative.

The quotient must be at least two digits in length. One position is required for the sign and one for the field mark (flag bit).

Examples

1. The macro-instruction
   DIV A, B, 99, 96
   will perform the division for $\bar{0}273\overline{)\bar{3}972}$ and store the result $\bar{0}014$ in storage location 00092 through 00095.
2. The macro-instruction
   DIV A, B, 96, 93
   will perform the division for $\bar{0}273\overline{)\bar{3}972.000}$ and store the result $\bar{0}014.549$ in storage locations 00089 through 00095.

NOTE: In both examples (1 and 2), A represents the address of the dividend $\bar{3}972$ and B represents the address of the divisor $\bar{0}273$.

Incorrect Positioning of Divisor

The following error conditions are caused by an incorrect B1 address.

Overflow

An incorrectly positioned divisor can cause more than nine successful subtractions and an incorrect quotient. The divide operation is terminated, the Overflow indicator and Overflow Arithmetic Check light are turned on, but processing will not stop unless the Overflow Check switch is set to STOP.

Loss of one or more high-order digits of the dividend

The high-order digit of the dividend is assumed by the 1620 to be one position to the left of the high-order digit of the divisor. The high-order digits of the dividend are lost if the divisor is positioned too far to the right. Processing continues with no indication of an incorrect quotient.

Incorrect termination (for subroutines which use the automatic divide feature)

If the B address is less than 10000, i.e., between 00100 and 09999, the divide operation will terminate when a subtraction occurs at 0XX99. This, in effect, restricts the size of the dividend to 10,020 digits if only 20,000 positions of core storage are installed.

Average Execution Time

Fixed length and variable length with automatic divide

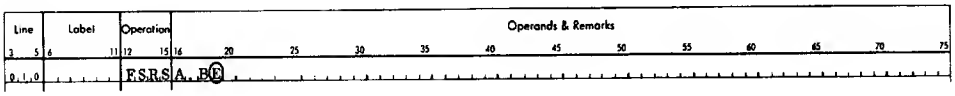
$$\text{Average time (in ms)} = 980 + .040\,\text{LDVD} + (.520\,\text{LDVR} + .740)\,(100 - \text{B1})$$

where LDVD is length of the dividend field,
LDVR is length of the divisor field, and
B1 is value specified in macro-instruction.

NOTE: Multiply 3.2 times the result to find the average execution time for the fixed length and variable length subroutines without automatic divide.

Floating Shift Right

Macro-instruction



FSRS macro

The effect of this macro-instruction is to shrink the mantissa by shifting it to the right and truncating the low-order digits. The A address is normally the units position of the mantissa.

$$\bar{M}MMMMMMM\bar{E}E$$

units position
of mantissa

The B address specifies the digit of the mantissa which will become the low-order digit of the mantissa.

Floating shift right (contd.)

Operation

The field at the B address (the portion of the mantissa to be retained) is shifted right to the location specified by the A address. The exponent is not moved or altered. For example, the macro-instruction

```
FSRS    00097,00093
```

causes the mantissa

$\bar{3}0590011325\bar{7}\bar{0}1$

Storage Address 00093	Storage Address 00097

to be shifted, producing the following result

$0000\bar{3}059001\bar{1}\bar{0}1$

Storage Address 00093	Storage Address 00097

Vacated high-order positions are set to zeros. An existing flag at the A address is retained for algebraic sign; the field flag bit is transmitted with the high-order digit of the B field.

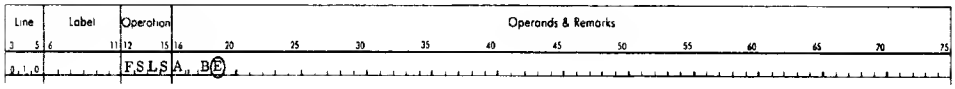
Average Execution Time

Fixed length and variable length

Average time (in μs) = 4960 + 960L − 880 (A−B)

Floating Shift Left

Macro-instruction



FSLS macro

The effect of this macro-instruction is to expand the mantissa by shifting it to the left and filling the vacated positions with zeros. It is important to note that the B address is the low-order position of the field moved, and the A address is the high-order position of the resulting field.

Floating shift left (contd.)

Operation

The field at the B address, which is the *low-order* digit of the mantissa, is shifted left so that the *high-order* digit is moved to the location specified by the A address. The exponent is not moved or altered. For example, the macro-instruction:

```
FSLS 00090, 00097 (E)
```

causes the mantissa

```
00̄11325̄7̄0̄1
↑     ↑
Storage  Storage
Address  Address
00090    00097
```

to be shifted, producing the following result

```
1̄132570̄0̄1
↑     ↑
Storage  Storage
Address  Address
00090    00097
```

An existing flag bit at the Q address is retained for algebraic sign; the field flag bit is transmitted with the high-order digit of the Q field.

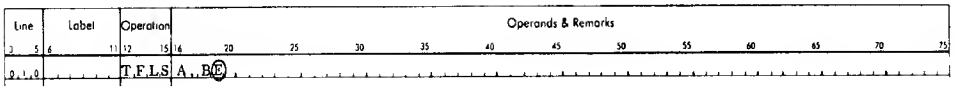
Average Execution Time

Fixed length and variable length

Average time (in μs) = 6460 + 1520 (B — A) — 360L

Transmit Floating

Macro-instruction



TFLS macro

The B address refers to the low-order digit of the floating point field exponent, whereas the A address refers to the low-order position to which the field is transmitted.

Operation

The field at the B address is transmitted to the location specified by the A address. The B field remains unchanged in storage. Flag bits in the three low-order positions of the B field are also transmitted; starting with the fourth low-order position only one additional flag bit is transmitted, and it stops transmission. For the variable length subroutine, L must be $\leq$49, where L equals the number of mantissa digits in the field to be transmitted. For the fixed length subroutine, L must be $\leq$ 19.

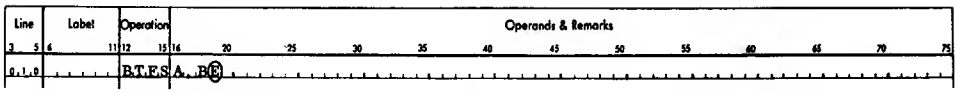
Average Execution Time

Fixed length and variable length

Average time (in μs) = 400 + 40L

Branch and Transmit Floating

Macro-instruction



BTFS macro

The B address is normally the low-order position of the floating point field exponent, whereas the A address is the leftmost position of the next instruction to be executed.

Operation

The address of the next instruction is saved at a storage location equivalent to BTFS + 78 and the field at the B address is transmitted to the A address minus one. The normal exit of a routine which is entered by a BTFS is a branch back (BB) instruction. The instruction at the A address is the next one executed. The B field remains unchanged in core storage. Any flag bits in the three low-order positions of the B field are transmitted; starting with the fourth low-order position only one additional flag bit is transmitted, and it stops transmission.

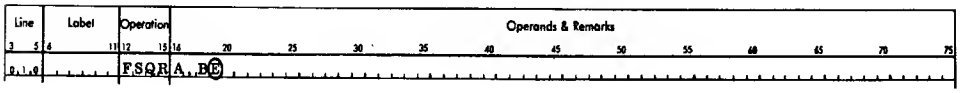
Average Execution Time

Fixed length and variable length

Average time (in μs) = 2280 + 40L

Floating Square Root

Macro-instruction



FSQR macro

The A and B addresses refer to the units position of the exponents of the fields.

Operation

The square root of argument B is extracted and the result, in floating point form, is stored at A. The argument, which must be in floating point form, is unchanged by the operation.

The floating point square root subroutine accepts all numbers within the floating point range that are greater than or equal to zero. If the argument is less than zero, the subroutine executes a programmed halt. The operator has two options:

1. Using the information found in the halt instruction, he may branch back to the main routine, or
2. Continue the execution of the subroutine and compute the square root of B .

Average Execution Time

Fixed length

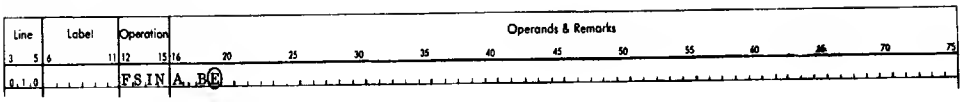
Average time = 120 ms

Variable length

Average time (in μs) = $620L^2 + 9776L + 5328$

Floating Sine

Macro-instruction



FSIN macro

The A and B addresses refer to the units position of the exponents of the fields.

Operation

The sine of argument B is computed and the result, in floating point form, is stored at A. The argument must be in radians and in floating point form. The computation does not disturb the original value of the argument.

The floating point sine subroutine accepts all numbers of floating range up to and including exponent $\bar{0}8$ (fixed length mantissa) or L (variable length mantissa). The operator may branch back to the mainline program as explained under SUBROUTINE ERROR MESSAGES.

For arguments with exponents less than $\bar{0}3$, the magnitude of the maximum truncation error in the mantissa of the result does not exceed 10^{-L}. Accuracy in the mantissa of the result decreases as the size of the argument (exponent of $\bar{0}3$ or greater) increases. The operator has the option of branching back to the main program or proceeding with the computation. Any result so computed will contain an error that varies directly with the magnitude of the exponent.

Average Execution Time (arguments less than 2π)

Fixed length

Average time = 150 ms

Variable length

With automatic divide

Average time (in μs) = $168L^3 + 3792L^2 + 13340L + 4708$

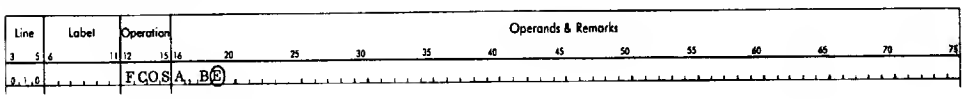
Without automatic divide

Average time (in μs) = $1.9\,[168L^3 + 3792L^2 + 13340L + 4708]$

NOTE: For all subroutines of this type, arguments greater than 2π are reduced by subtractions of 2π until within range. Therefore, the time required to perform these subtractions should be added to the average time required for an argument less than 2π.

Floating Cosine

Macro-instruction



FCOS macro

The A and B addresses refer to the units position of the exponents of the fields.

Operation

The cosine of argument B is computed and the result, in floating point form, is stored at A. The argument must be in radians and in floating point form. The computation does not disturb the original value of the argument.

The allowable range of the argument, maximum accuracy, etc., for the cosine subroutine are the same as those previously described for the sine subroutine.

Floating cosine (contd.)

Average Execution Time

Fixed length

Average time = 155 ms

Variable length

With automatic divide

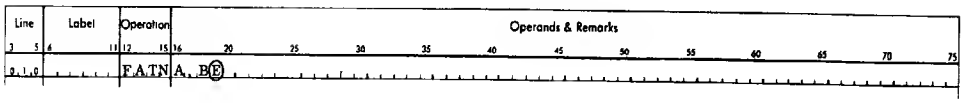
Average time (in μs) = $168L^3 + 3792L^2 + 13420L + 5228$

Without automatic divide

Average time (in μs) = $1.9\,[168L^3 + 3792L^2 + 13420L + 5228]$

Floating Arctangent

Macro-instruction



FATN macro

The A and B addresses refer to the units position of the exponents of the fields.

Operation

The floating point value of the arctangent of B is computed and the result stored at A. The argument must be in floating point form; the result in radians will be in floating point form.

The arctangent subroutine accepts any number within the floating point range.

During the evaluation of the arctangent of B, use will be made of the divide subroutine.

The maximum truncation error in the mantissa of the result is $\pm 10^{-L}$, except for results having an exponent less than or equal to $\bar{0}\bar{2}$ ($E \leq \bar{0}\bar{2}$). The maximum error for these results is ± 1 in the $(L + 1)$th decimal place. L = 08 for the fixed length mantissa.

Average Execution Time

Fixed length

Average time = 260 ms

Variable length

With automatic divide

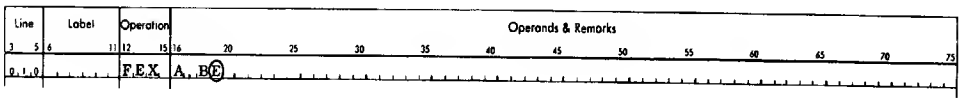
Average time (in μs) = $168L^3 + 2996L^2 + 7792L + 7260$

Without automatic divide

Average time (in μs) = $1.9\,[168L^3 + 2996L^2 + 7792L + 7260]$

Floating Exponential (Natural)

Macro-instruction



FEX macro

Operation

The A and B addresses refer to the units position of the exponents of the fields. The value of e^B, where B is in floating point form, is computed and the result, also in floating point form, is stored at A.

An input value that exceeds

227.955924206n.....n ($\bar{2}27955924206n.....n\bar{0}3$)

causes an exponent overflow and one which is less than

Floating exponential (natural, contd.)

$-227.955924206n \ldots\ldots n$ ($\bar{2}27955924206n \ldots\ldots \bar{n}\bar{0}3$)
causes an exponent underflow. An exponent overflow or underflow causes the subroutine to examine core storage position 401 to determine the course of action.

For negative arguments, the subroutine uses the absolute value of the argument to evaluate the function, and then finds the reciprocal value.

For positive and negative arguments, the maximum truncation error in the mantissa of the result is $\pm 10^{-L}$.

Average Execution Time

Fixed length

Average time = 160 ms

NOTE: Add 70 to the average time if B is negative .

Variable length

With automatic divide

Average time (in μs) = $168L^3 + 35824L^2 + 15890L + 26418$

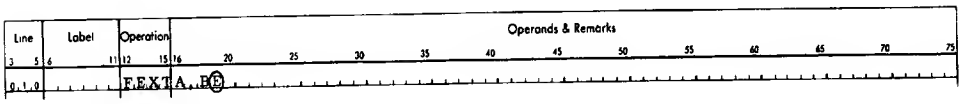
Without automatic divide

Average time (in μs) = $1.9[168L^3 + 35824L^2 + 15890L + 26418]$

NOTE: For a negative argument, add the result of $520L^2 + 1880L + 1480$ to the average time.

Floating Exponential (Base 10)

Macro-instruction



FEXT macro

The A and B addresses refer to the units position of the exponents of the fields.

Operation

The value of 10^B, where B is in floating point form, is computed and the result, also in floating point form, is stored at A.

An input value which exceeds $98.9n \ldots\ldots n$ ($\bar{9}89n \ldots\ldots n\bar{0}\bar{2}$) causes an exponent overflow and one which is less than $-98.9n \ldots\ldots n$ ($\bar{9}89n \ldots\ldots n\bar{0}\bar{2}$) causes an exponent underflow. An exponent overflow or underflow causes the subroutine to examine core storage position 401 to determine the next course of action.

This subroutine handles negative arguments in the manner they are handled by the natural exponential subroutine. Maximum accuracy is the same.

Average Execution Time

Fixed length

Average time = 145 ms

NOTE: Add 70 ms to the average time if B is negative.

Variable length

With automatic divide

Average time (in μs) = $168L^3 + 3656L^2 + 15414L + 24538$

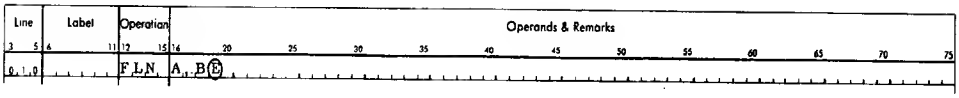
Without automatic divide

Average time (in μs) = $1.9\ [168L^3 + 3656L^2 + 15414L + 24538]$

NOTE: For a negative argument, add the result of $520L^2 + 1889L + 1480$ to the average time.

Floating Logarithm (Natural)

Macro-instruction



FLN macro

The A and B addresses refer to the units position of the exponents of the fields.

Operation

The floating point value of the ln B is computed and stored at A. Input arguments must be in floating point form.

This subroutine accepts all arguments greater than zero within the floating point range. An input argument equal to or less than zero results in a programmed halt. A branch back to the main program can be effected as described under SUBROUTINE ERROR MESSAGES.

Average Execution Time

Fixed length

Average time = 290 ms

Variable length

With automatic divide

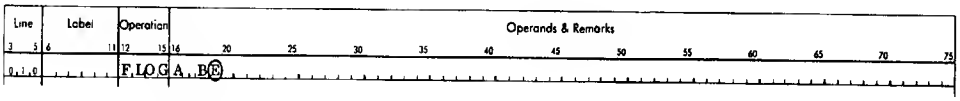
Average time (in μs) = $168L^3 + 3440L^2 + 10530L + 12180$

Without automatic divide

Average time (in μs) = $1.9\,[168L^3 + 3440L^2 + 10530L + 12180]$

Floating Logarithm (Base 10)

Macro-instruction



FLOG macro

The A and B addresses refer to the units position of the exponents of the fields.

Operation

The floating point value of the $\log_{10}B$ is computed and stored at A. Input arguments must be in floating point form.

This subroutine accepts all arguments greater than zero within the floating point range. An input argument equal to or less than zero results in a programmed halt. A branch back to the main program can be effected as described under SUBROUTINE ERROR MESSAGES.

Average Execution Time

Fixed length

Average time = 305 ms

Variable length

With automatic divide

Average time (in μs) = $168L^3 + 3608L^2 + 11610L + 15108$

Without automatic divide

Average time (in μs) = $1.9\,[168L^3 + 3608L^2 + 11610L + 15108]$

Maximum, twelve subroutines

The user may add from one to twelve subroutines to the floating point subroutine package for card or tape. Each new subroutine may use whatever number of operands is specified by the programmer. The minimum number of operands allowed is two; however, both the A and B operands may be the same.

Procedure

To add a subroutine, it is necessary to:

1. Insert library change card in operation code table of the SPS processor deck.
2. Write the subroutine in SPS language with origin at 5000 (DORG 5000), keeping in mind certain factors regarding PICK, mantissa length (L), and modifications with regard to the subroutine itself.
3. Assemble and condense the subroutine.
4. Discard the first two loader cards and the last seven table cards of the condensed object deck.
5. Prepare a header and a trailer card for the new subroutine deck, placing the header card in front of the new deck and the trailer card behind it.
6. Insert the new subroutine in the existing subroutine deck.

The procedure here described is for adding card subroutines; however, the final part of this section describes the additional steps required to convert the added subroutines to paper tape.

Inserting the Library Change Card

Library change card

For each macro-instruction added, a library change card must be prepared for insertion in the processor deck. These cards must be inserted immediately in front of the last nine cards of the processor deck.

Unique mnemonic operation code

The programmer must assign a unique mnemonic operation code to each new macro-instruction. The code may consist of from one to four alphameric characters. Any of the following four codes, for example, would be acceptable.

X
MT
MTX
MTRX

The mnemonic operation codes are punched into the library change cards, using core storage alphameric character coding (2 positions for each character). MTRX is punched as $\bar{5}4635967$. Note that a flag must be punched over the leading digit.

Format of library change card

The format of the library change card is as follows:

Columns	1-8	mnemonic operation code in alphameric character coding (left-justified in the field).
	9-10	sequence number of subroutine, starting with 18.
	11	$\bar{7}$.
	12	record mark (0, 2, 8 punches).
	13-62	blanks.
	63-64	$\bar{0}1$.
	65-69	address of leftmost location where data from columns 1-11 is to be stored. Note that column 65 must contain a flag.

Format of library change card (contd.)

Columns 70-74 (contd.) address plus 1 of rightmost location where data from columns 1-11 is to be stored. Note that column 70 must contain a flag.
75 blank.
76 ¯ (flag only).

The address in card columns 65-69 for subroutine 18 (next number after 17 Library subroutines) is 17374 and is increased by eleven for each subsequent subroutine. The address for tape SPS is 16156. The address in card column 70-74 is 17385, the same address as is used in 65-69, with an additional constant eleven added to compensate for the rightmost address. This address is 16167 for tape SPS.

Writing the Subroutine

When writing the subroutine, the programmer should be aware of the linkages that are generated by macros. Suppose the Floating Subtract macro-instruction

```
FS   A, B
```

Linkage

is used. The following linkage will be generated in the mainline program:

```
TFM     PICK + 11, * + 23
B       SUBR
DORG    * - 4
DSA     A, B
```

The branch is to the secondary linkage for the specific subroutine used (in this case, Floating Subtract). The secondary linkage is generated by the subroutine processor at the time the subroutines are being relocated and punched into the object deck. Examples of secondary linkage are:

Secondary linkage

```
1. SUBR   TFM    PICK + 402, ADDR
          B      PICK
2. SUBR   TFM    PICK + 402, ADDR
          B      PICK + 104
```

where ADDR is the address of the first instruction of the subroutine in question.

This linkage moves the starting address of the subroutine (Floating Subtract) into PICK to allow a later branch to the subroutine. Next, the secondary linkage branches to PICK. PICK moves the data contained in the A operand into Alpha and the B operand data into Beta. It also computes the return address to the mainline program and branches to the subroutine. After the subroutine is executed, the program branches back to the pick subroutine.

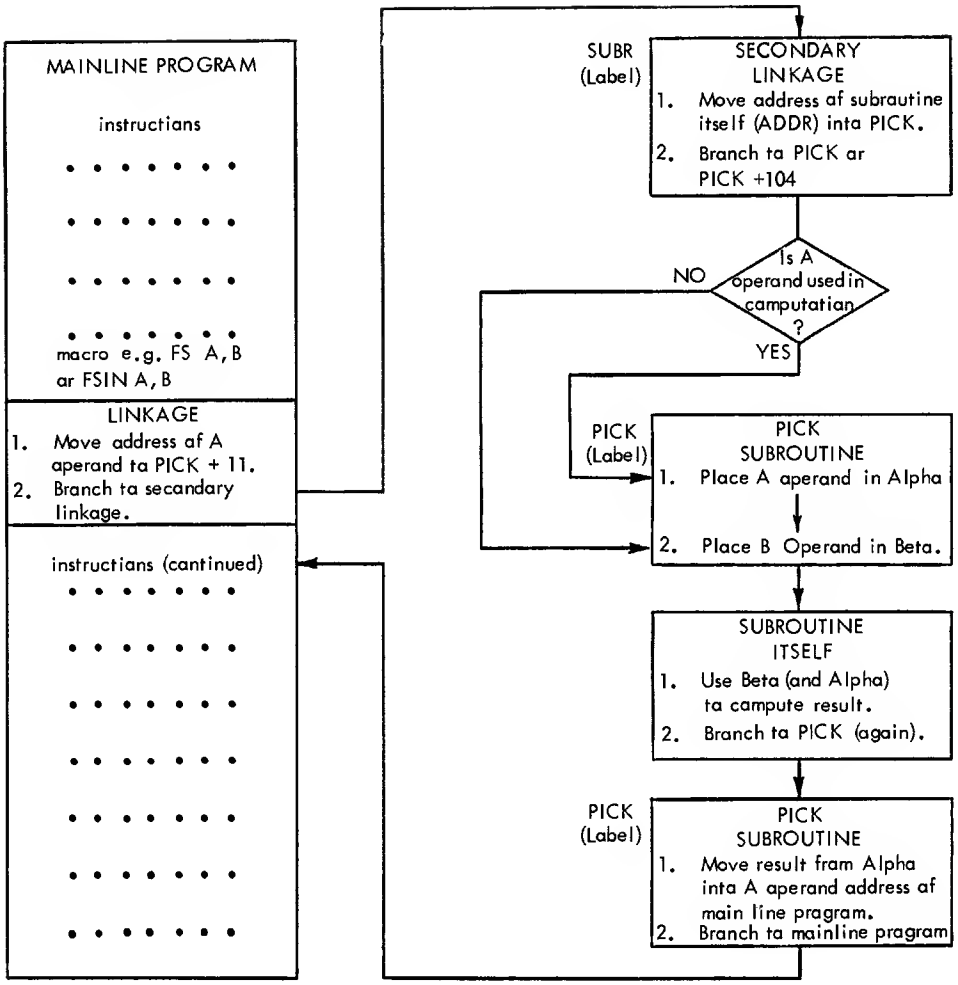
Sequence of events initiated by macro

The block diagram on the opposite page shows the sequence of events that occurs when the macro-instruction equivalence is encountered during execution of the object program.

NOTE: When the A operand in the diagram is used in the computation, the secondary linkage branches to PICK, and when the A operand is not used in the computation (as in the functional subroutines), the secondary linkage branches to PICK + 104, and the B operand alone is placed in Beta.

Provision necessary for more than two operands

Note also that if the macro-instruction contains more than two operands, arrangements must be made in the programmer's subroutine to store the B operand in a location other than Beta, because when the program re-enters the PICK subroutine to pick up the C operand, PICK will automatically store it in Beta. The C operand should be stored in the same manner if the program is to return to pick up a fourth operand, because PICK will place the D operand in Beta.



Listed below are certain PICK address equivalents for fixed length and variable length subroutine decks, as well as for the subroutine deck which uses automatic floating point. The subroutine writer should be familiar with them to make use of PICK when writing his subroutine.

PICK address equivalents

ADDRESS EQUIVALENTS			DESCRIPTION
FIXED LENGTH	VARIABLE LENGTH	AUTOMATIC FLOATING POINT	
PICK	PICK	PICK	Entry for subroutines that *use* A operand data in the computation.
PICK + 104	PICK + 104	PICK + 24	Entry for subroutines that *do not use* A operand data in the computation.
PICK + 140	PICK + 140	PICK + 60	Re-entry to pick up additional operand (other than the A and B operands).
PICK + 414	PICK + 402	PICK + 174	Address of subroutine (P address of instruction that branches to the subroutine).
PICK + 416	PICK + 404	PICK + 176	Re-entry from subroutines to store result of computation.
PICK + 482	PICK + 434	PICK + 194	Return address of mainline program (P address of instruction that branches to mainline program).
PICK + 711	PICK + 657	PICK + 417	Alpha (A operand data itself).
PICK + 743	PICK + 802	PICK + 562	Beta (B operand data itself).

Working areas and constants shared

There are various working areas for constants in the pick subroutine that may be used (shared) by the added subroutines. The programmer may refer to the subroutine program listing (provided with library package) to make effective use of the pick subroutine.

The return address to the mainline program (PICK + 482 for fixed length subroutines or PICK + 434 for variable length subroutines) calculated by the PICK subroutine is only correct if all operands associated with the subroutine have been processed.

Results stored in Alpha

The computed result is always assumed to be stored in Alpha (PICK + 711 for fixed length subroutines or PICK + 657 for variable length subroutines). In addition, the result at Alpha is stored by the pick subroutine at the address specified by the A operand, prior to return to the mainline program.

When writing the subroutine:

Programmer sets flags

1. A flag must be set over position O_0 and/or O_1 of instructions where the P and/or Q operands are relative to the origin 05000, e.g., an instruction located at 05300, such as

   TF * + 23, * −1

   in machine language should be $\bar{2}\bar{6}0532305299$, and the instruction should be written TF * + 23, * — 1, 01. If the P operand alone were relative, then only O_0 would be flagged, as

   AM * + 18, 5, 07

Programmer utilizes the pick subroutine

2. Since PICK is a sub-subroutine common to all subroutines in the subroutine package, except DIV, FSRS and FSLS, it is to the advantage of the subroutine writer to make use of it. The listing of the appropriate pick subroutine (furnished with the library package) should be studied. Briefly, PICK performs the following operations. It

Functions of PICK

   a. Moves the A operand into Alpha (exponent and mantissa).
   b. Moves the B operand into Beta (exponent and mantissa).
   c. Calculates the return address to the mainline program.
   d. Stores the computed result (which is in Alpha) back into the address of the A operand.
   e. Contains constants and storage areas that are common to other subroutines in the package.

The programmer will use instructions which make reference to the PICK subroutine (to both PICK instructions and constants) in his subroutine. The subroutine relocator program must adjust these addresses to make them correspond to the actual addresses of PICK in the object program. This adjustment is made by using a pseudo constant (DC statement). The constant does not become part of the object program; its only function is to indicate to the subroutine relocator program that the instructions which follow are to be modified.

One statement can modify up to 25 instructions. Each instruction, whether or not it is to be modified, requires two digits in the pseudo constant, one for the P operand and one for the Q operand. The statement itself consists of three operands: the first specifies the length of the constant which may not be greater than 50 nor less than 2; the second, the actual constant; the third, the storage address of the constant. This address must be specified as an absolute value in the following form: 00320 for a 20-digit constant, 00342 for a 42-digit constant, etc. The P and Q operand modifier constants follow.

Operand modifiers

P AND Q OPERAND MODIFIERS	MODIFICATION
0	No modification
1	Add L
2	Subtract L
3	Add 2L
4	Subtract 2L
5	Modify with respect to PICK, no L modification
6	Modify with respect to PICK, add L
7	Modify with respect to PICK, subtract L
8	Modify with respect to PICK, add 2L
9	Modify with respect to PICK, subtract 2L

Example of modification

The following example shows how a variable length mantissa subroutine may be modified, by use of modifier constants, to use three operands in its computation. Secondary linkage 1 is used in this example.

The A operand data is stored in Alpha (PICK + 657), and the B operand data is in Beta (PICK + 802). Therefore:

```
      DC    6, 275050, 306
SUBR  TR    GAMMA-1, 801, 0      Transmit Beta into Gamma
      TFM   402, *+20, 17        Set up return address to added subroutine
      B     140                  Go to PICK subroutine to obtain next operand
```

NOTE: Intervening DORG statements and constants between instructions are never modified in this manner.

Data from the last operand processed will be Beta (PICK + 802). The maximum number of operands allowed in secondary linkage 1 is two; however, the A and B operands may be the same.

Incorporating New Subroutines in the Subroutine Deck

Insert subroutine

After a subroutine is written, it is assembled and condensed; the first two and last seven cards are removed from the object deck and discarded. These cards are replaced by a subroutine header card and subroutine trailer card. The header card precedes the subroutine and is punched according to the following format.

Header and trailer cards

Preparation of Header and Trailer Cards

Header card format

Columns 1-4 Length of subroutine — 4 digits in the form $\bar{x}xxx$

5-13 Subroutine numbers (each in two digits $\bar{x}x$) followed by a record mark; a subroutine may have up to four entrances represented by four different macro-instructions.

Header card format (contd.)

Columns 14-23 (contd.) Number of storage positions (three digits x̄xx), between the second (third and fourth) entrance(s) and the regular entrance of the subroutine. These 3-digit fields must be terminated by a record mark (for example, 1̄20186=). If the subroutine has only one entrance, a record mark should be punched in column 14.

24 0 (zero) for subroutine deck without automatic divide; 1 for subroutine deck with automatic divide; = for variable length subroutine decks with divide or with automatic floating point; 0̄ for variable length subroutines without automatic divide.

25-43 The secondary linkage in machine language. The linkage contains two instructions, the second of which is always a branch (operation code 49). The operation code in the first instruction and the three addresses can be specified by the programmer. The breakdown of these columns is as follows:

25-26 Two-digit numerical operation code for the first instruction. Modification to the P and Q addresses is indicated by a flag or no flag on the first and second digit, respectively. A flag implies that the address is relative to the subroutine itself while no flag means it is relative to the pick subroutine.

27-31 P address of the first instruction, expressed as an increment to PICK or the subroutine. For example, PICK + 23 will be 00023 and SUBR 1 + 59 will be 00059.

32-36 Q address of the first instruction, expressed as an increment to PICK or the subroutine. For example, PICK + 23 will be 00023 and SUBR 1 + 59 will be 00059.

37-38 Operation code 49; a flag or no flag on digit 4 indicates modification to the P address with respect to the subroutine or PICK.

39-43 P address expressed as an increment to PICK or the subroutine.

44-75 Blanks.

76 0 (zero).

77-80 Sequence number (see SEQUENCE NUMBERING OF SUBROUTINES).

Trailer card format

The subroutine is followed by a trailer card punched with a 1 in column 76 (blanks in other columns).

Inserting new subroutine in subroutine deck

The new subroutine, together with its header and trailer cards, is inserted into the subroutine deck in front of the last card, which is identified by a record mark (=) in column 76.

The SPS processor restoration deck which restores that part of the SPS main processor destroyed by the subroutine processor program is behind the card with the record mark (=) in column 76.

Bypassing PICK

Alternative to PICK

The subroutine writer may, if he desires, bypass PICK completely. He accomplishes this by setting up the secondary linkage on his header card (columns 25-43). Since the first linkage puts the address of the A operand in PICK + 11, the secondary linkage can move it from there to any place in the subroutine itself (and branch to the subroutine). This is what is done in the DIV, FSLS, and FSRS subroutines, where the secondary linkage is of the form

```
TF   ADDR + 11, PICK + 11, 0
B    ADDR, , 0
```

Sample Problem Illustrating Header Card, Subroutine, and Trailer Card

In this example the subroutine is to be inserted in the variable length subroutine deck without automatic divide. The Floating Branch and Transmit subroutine (BTFS) is used as the new subroutine, thus it is assumed that the subroutine deck contains no BTFS subroutine. These examples are intended to show the header card coding, the modifier constants, the O_0 O_1 flag indicators associated with the new subroutine, and the trailer card coding. For each field of the header and trailer cards, the data contained in the field as well as a description of the data is given.

Header card

Columns 1-4 :	$\bar{0}079$		Program requires 79 storage positions.
5-13:	$\bar{1}7\,\bar{+}$		Subroutine identifying number.
14-23:	$\bar{+}$		Subroutine has only one entry point.
24:	$\bar{0}$		Variable length subroutine without automatic divide.
25-43:	$1\bar{6}$ 00402	00000 49 00104	These instructions correspond to the secondary linkage: TFM PICK + 402, ADDR B PICK + 104 The P operands of the TFM and B will be modified by adding the address of PICK when it is found. The Q operand of the TFM will be modified by adding the starting address of the BTFS subroutine to it.
44-75:	Blanks		
76:	$\bar{0}$		
77-80:	$\bar{7}\bar{0}00$		The flagged 7 is the identifying number of the subroutine. The $\bar{0}00$ is the card sequence number.

Subroutine

	DORG	5000	Standard DORG statement for all subroutines.
	DC	14, 05000005050500, 314	This statement provides the modifier digits for the seven instructions which follow.
BTFS1	TF	* + 66, STORE + 6, 0	A flag over O_0 indicates to the subroutine processor that the P operand must be modified with respect to the relocated addresses of the BTFS subroutine. With respect to PICK, the Q operand is modified by the second digit of the pseudo constant.
	TF	* + 30, * + 54, 01	The P and Q operands need only be modified with respect to the BTFS subroutine. Therefore, the only modification required is flagging O_0 and O_1.
	SM	* + 18, 3, 010	* + 18 is modified with respect to its own subroutine. The Q operand needs no modification since $\bar{0}3$ is wanted. The flag on Q_{10} is needed in the computation.
	TF	, BETA-2	With respect to PICK, the Q operand is modified by the eighth digit of the pseudo constant.
	TF	* + 30, STORE + 30, 0	O_0 flagged to modify * + 30 with respect to BTFS subroutine. Q operand was previously modified. With respect to PICK, the Q operand is modified by the tenth digit of the pseudo constant.
	BT	, BETA	With respect to PICK, the Q operand is modified by the twelfth digit of the pseudo constant.
	B		No modification.
	DEND		No modification.

Trailer card

Column 76	1

Adding a Subroutine to Tape SPS

The steps required by the user to add a subroutine to the subroutine tape and include his macro-instruction mnemonic in the operation code table of the processor are:

Procedure (summary)

1. Write the subroutine in SPS language with origin at 5000 (DORG 5000). Certain factors regarding PICK, mantissa length (L), and modification of the subroutine must be observed when writing the subroutine.
2. Assemble the subroutine.
3. Prepare the subroutine header data so that it may be entered at the keyboard when called for by the tape modifier program.
4. Load the tape modifier program.
5. Process the subroutine tape and subroutine to be added, making the changes directed by the typed messages. In this operation the modifier program assists the programmer in combining his subroutine with the existing subroutine to produce a new subroutine tape.
6. Using the modifier program, process the processor program and make the changes required to include the new macro-instruction (mnemonic) in the operation code table. This procedure is also directed by typewriter messages.

Preparing a New Subroutine Tape

The following procedure must be followed by the programmer in order to prepare the new subroutine tape.

Incorporating new subroutine in existing subroutine tape

Load the tape modifier program

1. Thread the modifier program.
2. Key in 36 00000 00300 at the typewriter.
3. Depress the release and start keys.

When the machine halts, the operator then:

1. Threads the existing subroutine tape.
2. Depresses the start key.

A message

Messages

SET PROGRAM SWITCHES AND DEPRESS START

will be typed. Program switch 3 should be turned on and all other program switches off. The start key should then be depressed.

NOTE: If the operator fails to set the switch prior to depressing the start key, the message

PROGRAM SWITCHES HAVE NOT BEEN SET

will be typed out. The operator may again proceed by setting the switch and depressing the start key.

After the subroutines have been copied onto the new tape, the message

MARK LAST RECORD READ, REMOVE MAIN SUBR TAPE,
NOW THREAD TAPE OF SUBR BEING ADDED

appears at the typewriter. The operation specified by the message should be executed and the start key depressed.

The message

ENTER HEADER DATA

is then typed out. If an error occurs while typing the header data, it can be corrected by manually branching to the previous Read Typewriter instruction. The address of this instruction may be determined by

1. Depressing the Release and Single Cycle keys, and
2. Subtracting 12 from the address which is displayed in the MAR register.

A card image of the header card data must be entered at the typewriter, with zeros substituted for blank columns. This header data is prepared in the same manner as it was for the card system (see HEADER CARD FORMAT). The release and start keys are depressed to resume processing and to complete copying of the new subroutine. The message

Messages

RETHREAD MAIN SUBROUTINE

is typed. The operator must be careful when rethreading the tape (previously marked) to back up the tape by one record and resume reading with the last record read, prior to removing the main subroutine tape in order to add the new subroutine tape. The start key is depressed. The machine stops with the reader no feed light turned on after the new subroutine tape is completed.

If another tape is to be modified, the operator may ready the program by
1. Turning program switch 4 on,
2. Depressing the reset and start keys, and
3. Turning program switch 4 off.

Adding Macro-Instruction to Processor Tape

Incorporating OP codes in processor tape

The following procedure describes how the tape modifier program is used to update the processor operation code table to produce a new processor tape.

If the modifier program is not in storage, it must be loaded.

To load the tape modifier program:
1. Thread the modifier program.
2. Key in 36 00000 00300 at the typewriter.
3. Depress the release and start keys.

The operator then:
1. Threads the SPS tape processor,
2. Turns program switch 2 on, and
3. Depresses the start key.

The message

Messages

ENTER ADDITIONS TO OP CODE TABLE ONE AT A TIME. THE LAST ENTRY SHOULD BE FOLLOWED BY A RECORD MARK.

will be typed. After each OP code entry of the new macro-instruction is entered at the typewriter, the release and start keys should be depressed. The data entered should consist of the following:

1. Mnemonic operation code in alphameric character coding (left justified in the field).
2. Sequence number of subroutine, starting with 18.
3. $\overline{7}$

The record mark which identifies the final mnemonic to be entered signals the modifier program to copy the processor program and insert the new mnemonic(s) in the OP code table. When this operation is completed, the program halts.

No more than a total of twelve operation codes can be added to the OP code table. If the operator tries to enter OP codes in excess of 12, the following message will be typed:

THE SPACE RESERVED FOR ADDITION OF OP CODES HAS BEEN FILLED. THE FOLLOWING HAS NOT BEEN INSERTED XXX XXX

where XXX represents an OP code that cannot be added to the table. The tape modifier will only add those OP codes up to twelve.

Switches for modifier program

Summary of Program Switches for the Tape Modifier Program

Turn on:

- Switch 1 to modify the two-pass processor for additional storage (see that section).
- Switch 2 to add OP codes to the processor OP code table.
- Switch 3 to add subroutines to the subroutine tape.
- Switch 4 to correct a typing error which is made while entering the memory size or an OP code. Depress release and start keys, turn switch off and re-enter data.

NOTE: It is possible to run with both switches 1 and 2 on. This operation allows the programmer to modify the processor for additional storage and at the same time to add OP codes.

Organization

The SPS processor, which is available in card or paper tape form, is a two-pass program. The source program, written in the language described, furnishes the input for both passes. The functions of pass 1 and pass 2 are listed below:

Pass 1

Functions of pass 1

1. Checks for valid mnemonic operation codes, Invalid operations are considered NOPS and are processed as such if program switch 2 is off.
2. Processes symbolic labels and prepares a table of the symbolic labels and their assigned addresses for use in the second pass.
3. Assigns storage positions to instructions, work areas, and constants.
4. Performs checking necessary to produce error messages.

Pass 2

Functions of pass 2

1. Processes operation codes. Converts mnemonic program operation codes to their corresponding 1620 machine language codes.
2. Processes operands according to the type of operation code. Looks up assigned addresses and symbolic operands in the symbolic table prepared during pass 1. Performs address adjustment, if necessary, to complete the operands. Sets flags in the assembled instruction, as specified by the flag indicator operand.
3. Types error messages for those statements that cannot be assembled properly.
4. Prepares the assembled output and lists the symbol table, if desired.

Storage Layout

Operation code table

The storage layout of the SPS processor is shown in Figure 3. The operation code table contains all valid mnemonic operation codes and their equivalent machine language codes. Any alterations to the processor will change the addresses shown in this figure.

Increasing size of symbol table

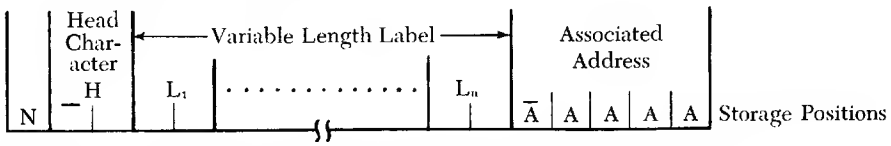
If additional storage (IBM 1623 Storage Unit) is available, it may be used to increase the size of the symbol table (shown in Figure 3) to accommodate a greater number of symbols. If the size of the symbol table is to be increased, the SPS processor program must be modified by the user. Modification techniques are explained under SPECIAL PROCEDURES FOR THE TWO-PASS PROCESSOR.

Symbol Table

Variable length

A variable length label entry is used to store as many labels as possible in the area reserved for the symbol table. Each label when stored takes the following form:

Format



where N = number of characters in label plus head character (2-6)
H = head character (two-position alphameric coding)
L.... L_n = five characters of label (two-position alphameric coding)
AAAAA = assigned address (five numerical positions)

NOTE: The rightmost position of L_n contains a flag for any true 6-character label.

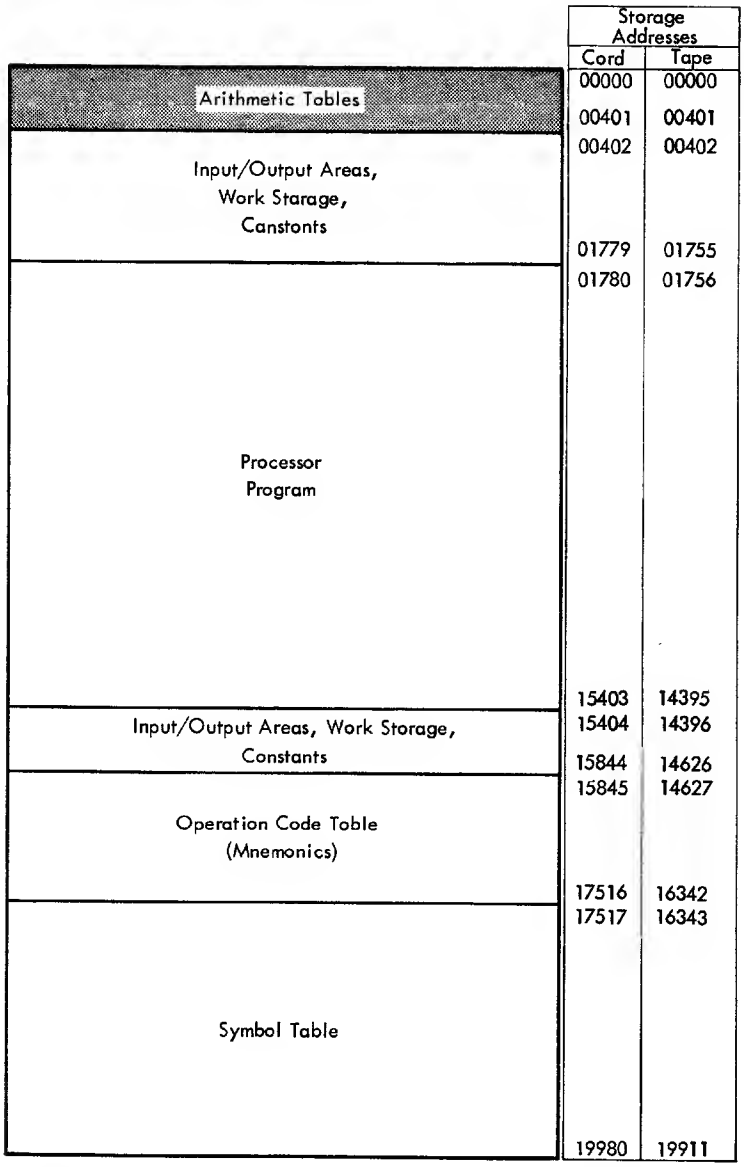


Figure 3. Storage Layout of 1620/1710 SPS Processor

Storage for minimum and maximum size labels

A label entry will always contain N, H, and A data and at least one L character. Therefore the minimum size label (one character) will require 10 storage positions. Each additional L character will use two additional storage positions up to eight positions. The maximum size label (5 or 6 characters) will require 18 storage positions.

Treatment of head character

A six-character label is stored without a head character and the leftmost character of that label occupies the head character (H) storage position. For the six-character label, a flag is placed in the rightmost position of L_n so that the processor may distinguish between a 5-character label with a head character and a true 6-character label without the head character. When a label which is not preceded by a HEAD statement is placed in storage, the head character (H) will be assigned as blank by the processor.

Capacity of label table

Because the maximum number of labels allowed cannot be specified due to

Formula for determining capacity

variable length symbols, the following formula may be applied to find the allowable number of symbols (within + 1 or — 1) for any given source program.

$$K = \left[\sum_{e=1}^{e=5} L_e \, (8 + 2e) \right] + 18L_6$$

where K = 19980 (standard capacity) minus address of symbol table (17517 card, 16343 tape)

e = number of characters in label

L_e = number of labels of length "e"

L_6 = number of six-character labels

NOTE: K should be increased by 20,000 or 40,000 when the processor is modified to accommodate 40,000 or 60,000 positions of storage, respectively.

Paper Tape Processor Program

Paper tape or typewriter input

The paper tape processor program accepts input for the first pass in either paper tape form or directly from the typewriter, depending upon the setting of the program switch. If the typewriter is used to enter the source statements, the processor produces a source program tape to be used as input for the second pass.

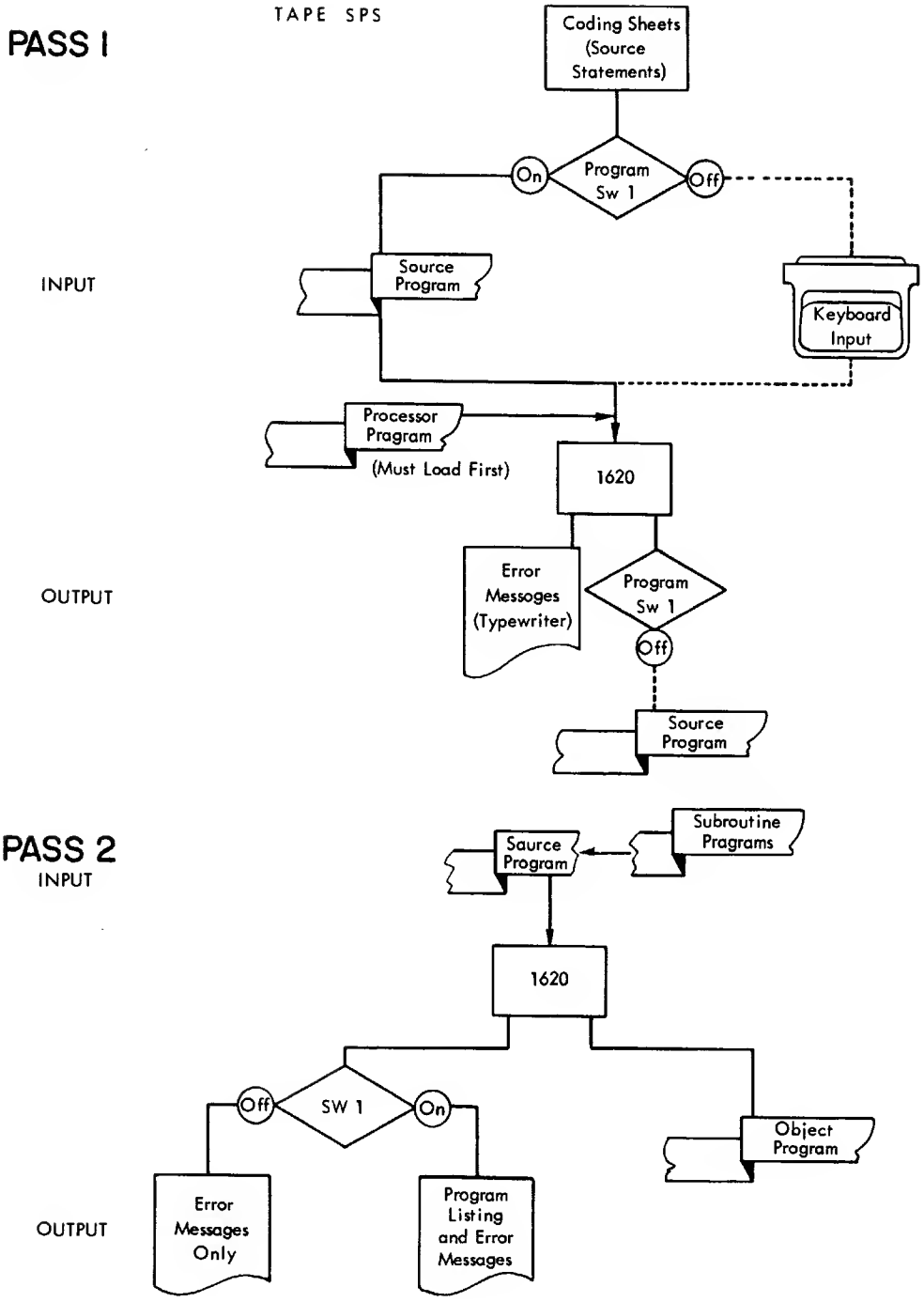
When subroutines are used in the source program, the subroutine program paper tape must follow the source program as input for the second pass.

Output from the second pass may include an object program tape and/or a typewriter listing. Error messages indicating errors in the source program are typed out during pass 1 or pass 2. The programmer has the option of correcting these errors either as indicated or after assembly is finished.

Format of Output Tape

Order of items on output tape

The output tape produced by pass 2 contains the assembled machine language instructions, constants, and other data that are part of the object program. Loading instructions appear at the beginning of the object tape followed by the object program, selected subroutines, and multiplication and addition tables (condensed form) in that order. A complete self-contained program tape is produced, ready to be entered in the 1620 or 1710.



Card Processor

Card or typewriter input

Input may be from cards or the typewriter. If the typewriter is used, a source program card deck is punched as output for pass 1. This card deck becomes the input for pass 2. Error messages are typed out for both passes. Affected statements may be corrected when the message is noted or at the completion of pass 2 after all messages have been recorded.

The input for pass 2 is the source program deck followed by the subroutines, provided they are used. The typewriter output from this pass may consist of the object program with error messages, or error messages only, as determined by the program switch setting (see Table 17).

Table 17. Operation of Program Switches for the Paper Tape Processor and the Card Processor for Passes 1 and 2.

SWITCH	PASS 1		PASS 2	
	ON	OFF	ON	OFF
1	For the paper tape processor, when input is from the paper tape reader. For the card processor, when input is from the card reader.	When input is from the typewriter.	When on, the entire input statement together with the assembled machine language instruction is typed out.	When off, no typeout of listing.
2	The machine stops after an error message has been typed, so that a corrected statement can be entered at the typewriter.	Processing continues after an error is typed, but the error is adjusted by the processor as indicated under ERROR CORRECTION.	Same as pass 1.	Same as pass 1.
3		For the card processor, switch must be off.	For the card processor, when the object program is to be in condensed form. (This switch should be on when pre-editing the source program.)	For the card processor, when the object program is to be in uncondensed form.
4	Turn on to correct a typing error made while entering a statement, and depress release and start keys, then off, and re-enter the entire statement at the typewriter.	Should be off when SPS processor is assembling data.	When on, no object program is punched except loader and arithmetic tables. (This switch should be on when pre-editing the source program.)	When off, the object program is punched.

Condensed or uncondensed output (2nd pass)

An object program card deck is produced in condensed form or uncondensed form, depending upon the setting of the program switches (see Table 17). The condensed card contains up to five machine language instructions, thus requiring fewer cards than the uncondensed version, which has multiple cards for each statement. Immediately after an uncondensed object deck is obtained from pass 2, the programmer may get a condensed deck by processing the source cards a third time (pass 3) as described under OPERATING PROCEDURES. If the programmer chooses to bypass the extra pass (pass 3), he may at some later time get a condensed deck from an uncondensed deck as described under CONDENSER PROGRAM.

Both contain loader and arithmetic tables

Both the condensed and uncondensed card decks are complete with loader and arithmetic tables. The uncondensed deck contains both symbolic and absolute information, but only absolute data is loaded.

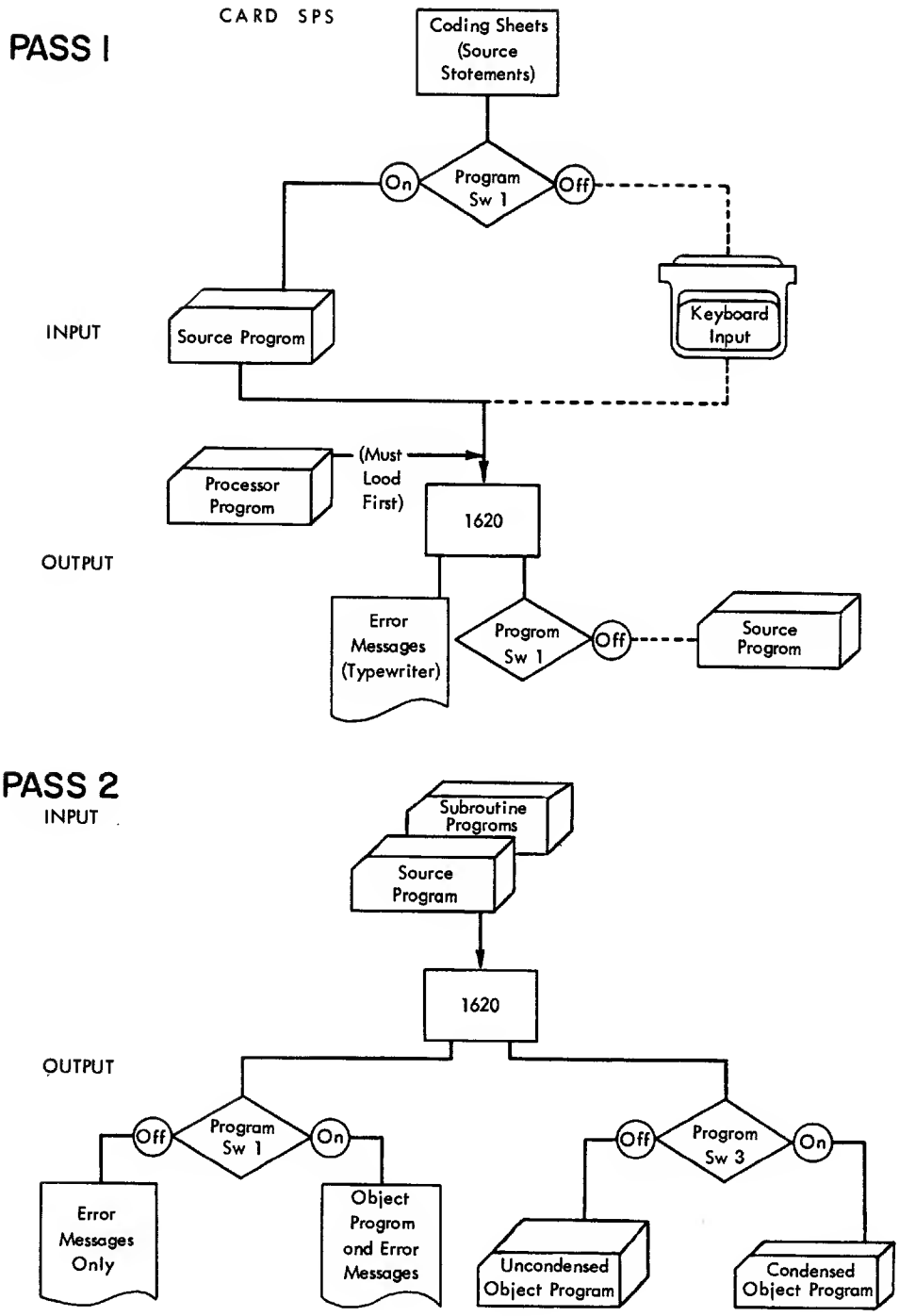
Format of Output Deck

The object program is preceded by two loader cards and followed by seven cards that perform the following:

1. Interrupt the loading sequence of the object program.
2. Load the arithmetic table.
3. Branch to the start of the object program or to halt.

Uncondensed Object Program Deck

To facilitate control panel wiring of a 407 for an off-line listing of the uncondensed deck, the individual card format of each statement is given. The number of cards per statement may range from one to several, depending on the type of operation (imperative, declarative, control, macro-instruction, comments) or type of individual statement. For the SEND or HEAD statements, no output cards are produced.



Imperative Operation Card Format

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns 1-5 page and line number.
6-10 high-order leftmost address where assembled instruction is to be stored.
11-22 assembled instruction.
23 $\pm$
63-64 $\overline{11}$.

Card 2. Columns (contd.)
- 65-69 leftmost address where assembled instruction is to be stored.
- 70-74 rightmost address plus one, where assembled instruction is to be stored.
- 76 $\overline{9}$.
- 77-80 card number.

Control Operation Card Format

DORG, DEND

DORG, DEND

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns
- 1-5 page and line number.
- 6-10 address specified.
- 76 9.
- 77-80 card number.

TRA

TRA

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns
- 1-5 page and line number.
- 6-10 leftmost address where instruction is to be stored.
- 11-22 assembled instruction (first instruction).
- 23 $\overline{=}$
- 63-64 $\overline{11}$.
- 65-69 leftmost address where instruction is to be stored.
- 70-74 rightmost address plus one, where instruction is to be stored.
- 76 $\overline{9}$.
- 77-80 card number.

Card 3. Same as card 2 (11-22 is second instruction).

TCD

TCD

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns
- 1-5 page and line number.
- 6-10 address specified.
- 76 9.
- 77-80 card number.

Cards 3-9. Arithmetic tables.

Cards 10-11. Loader program.

Declarative Operation Card Formats

DS, DSS

DS, DSS

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns
- 1-5 page and line number.
- 6-10 rightmost address of field.
- 13-17 field length.
- 76 9.
- 77-80 card number.

NOTE: For the DSS operation, columns 6-10 of card 2 contains the leftmost address of the field.

DAS

DAS

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns 1-5 page and line number.
6-10 leftmost address plus one, of field.
13-17 field length (number of alphameric characters).
76 9.
77-80 card number.

DSB

DSB

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns 1-5 page and line number.
6-10 rightmost address of first element of array.
13-17 element length.
76 9.
77-80 card number.

DSA

DSA

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. NOTE: A card of this type is punched for each operand.
Columns 1-5 page and line number.
6-10 rightmost address where field is to be stored.
13-17 field length (constant 00005).
18-22 the 5-digit field (address itself) being stored.
23 ≠
63-64 $\bar{1}8$.
65-69 leftmost address where field is to be stored.
70-74 rightmost address plus one, where field is to be stored.
76 $\bar{9}$.
77-80 card number.

DC, DSC

DC, DSC

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns 1-5 page and line number.
6-10 rightmost address where constant is to be stored.
13-17 field length of the constant.
76 9.
77-80 card number.

Card 3. Columns 1-5 page and line number.
6-n the constant itself starts in column 6 and is terminated by a record mark (≠) in the first column following the constant.
63-64 $\bar{0}6$.
65-69 leftmost address where constant is to be stored.
70-74 rightmost address plus one, where constant is to be stored.
76 $\bar{0}$.
77-80 card number.

NOTE: For the DSC statement, columns 6-10 of card 2 contain the leftmost address where the constant is to be stored.

DNB

DNB

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns
- 1-5 page and line number.
- 6-10 rightmost address where constant (blank) is to be stored.
- 13-17 field length.
- 76 9.
- 77-80 card number.

Card 3. Columns
- 1-5 page and line number.
- 7-n the numerical blanks (coded 4, 8) start in column 7 and are terminated by a record mark (≠) in the first column following the constant.
- 63-64 $\bar{0}7$.
- 65-69 leftmost address where constant (blanks) is to be stored.
- 70-74 rightmost address plus one, where constant is to be stored.
- 76 $\bar{0}$
- 77-80 card number.

DAC

DAC

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns
- 1-5 page and line number.
- 6-10 leftmost address plus one, where constant is to be stored.
- 13-17 field length (number of alphameric characters).
- 76 9.
- 77-80 card number.

Card 3. NOTE: A constant that contains over 25 characters causes two cards to be punched in this format. Up to 25 characters may be punched on each card.

Columns
- 1-5 page and line number.
- 6-n the constant itself starts in column 6 and is terminated by a record mark (≠) in the first column following the constant.
- 63-64 $\bar{0}6$.
- 65-69 leftmost address where constant is to be stored.
- 70-74 rightmost address plus one, where constant is to be stored.
- 76 $\bar{0}$.
- 77-80 card number.

Macro-instruction Operation Card Format

Card 1. Same as source statement card with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. Columns
- 1-5 page and line number.
- 6-10 leftmost address where first linkage instruction is to be stored.
- 11-22 assembled instruction.
- 23 ≠
- 63-64 $\bar{1}1$.
- 65-69 leftmost address where instruction is to be stored.

Card 2. (Contd.) Columns 70-74 rightmost address plus one, where instruction is to be, stored.
76 $\bar{9}$.
77-80 card number.

Card 3. Same as card 2 (second linkage instruction).

Card 4. NOTE: A card of this type is punched for each operand.
Columns 1-5 page and line number.
6-10 leftmost address where field is to be stored.
13-17 field length (constant 00005).
18-22 the address itself (5-position field) to be stored.
23 $\doteq$
63-64 $\bar{1}8$.
65-69 leftmost address where field is to be stored.
70-74 rightmost address plus one, where field is to be stored.
76 $\bar{9}$.
77-80 card number.

Comments Card Format

Card 1. Same as source statement with the exception of a 0 in column 76 and card number in columns 77-80.

Card 2. A digit 9 in column 76 and card number in columns 77-80.

Listing the Uncondensed Object Deck

Figures 4A and 4B show control panel wiring diagrams designed for listing an uncondensed object program on the IBM 407 and on the IBM 407-E8.

Condensed Object Deck

Condensed cards are punched as described under the particular card type.

Card Containing Instructions

Columns 1-12
13-24
25-36
37-48
49-60 five instructions.
61 $\doteq$ (record mark).
62 0.
63-64 $\bar{0}1$.
65-69 leftmost address where instructions are to be loaded.
70-74 rightmost address plus one, where instructions are to be loaded.
76 (flag only).
77-80 card number.

Card Containing Constants

Columns 1-61 constants may be from 1 to 60 characters followed immediately by a record mark ($\doteq$).
62 1.
63-64 $\bar{0}1$.
65-69 leftmost address where constants are to be loaded.
70-74 rightmost address plus one, where constants are to be loaded.
76 (flag only).
77-80 card number.

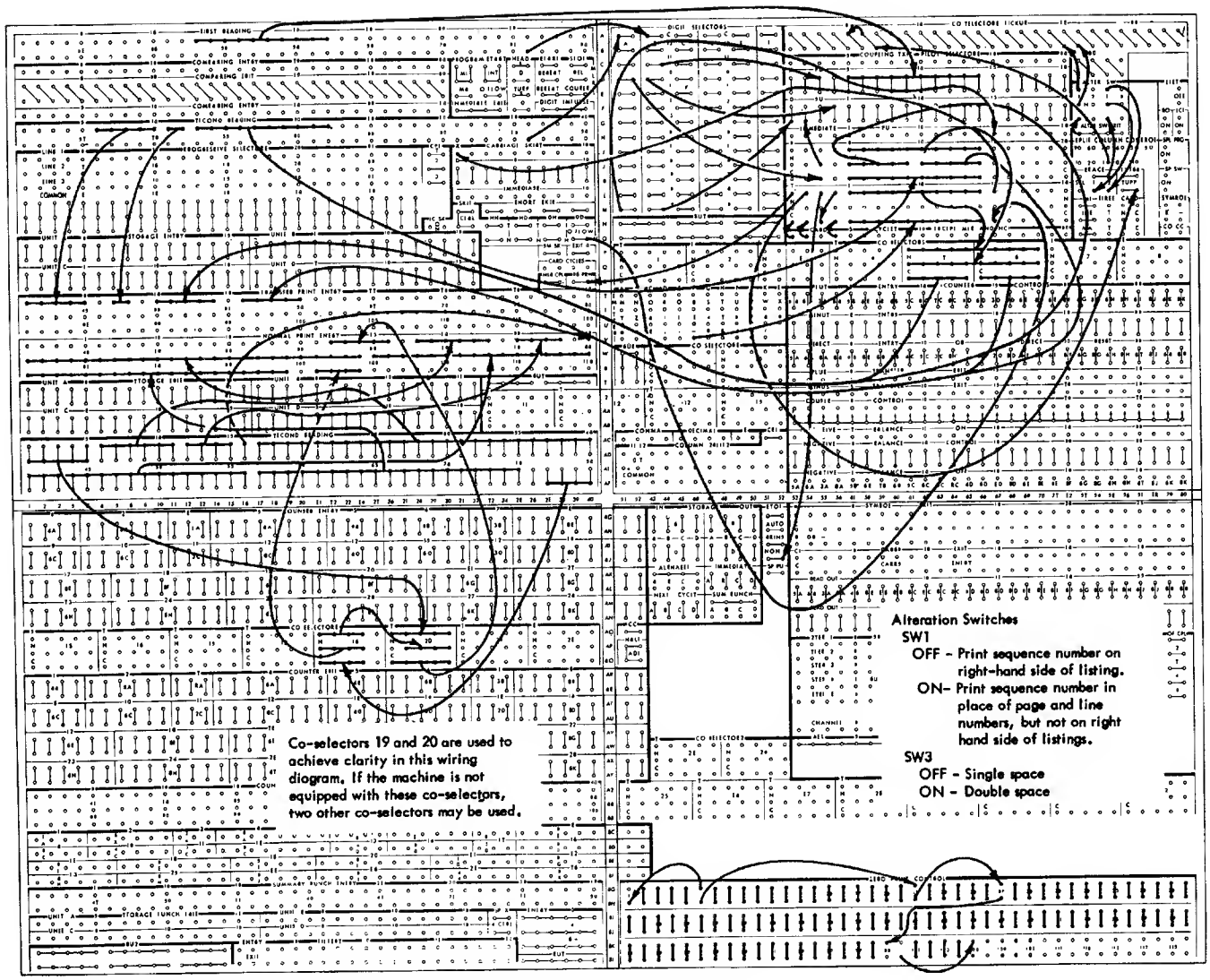


Figure 4A. 407 Control Panel Wiring Diagram for Listing Uncondensed Output

Program Switches

Switch settings, Table 17

Before processing the source statements, it is necessary to set the program switches to control processor functions, such as form of input used, action to be taken when an error is detected, and form of output generated. See Table 17 for an explanation of the operation of the program switches.

Error Messages

The error message codes that may be typed out on the typewriter during pass 1 and/or pass 2 of an assembly are listed in numerical sequence.

ERROR MESSAGE CODE	DESCRIPTION OF ERROR
ER1	A record mark is in the label or operation code field.
ER2	For address adjustment, a product greater than ten digits has resulted from a multiplication.
ER3	An invalid operation code has been used.
ER4	A dollar sign, which is being used as a HEAD indicator, is incorrectly positioned in an operand.

Error Messages (contd.)

ER5	a. The symbolic address contains more than six characters. b. The actual address contains more than five digits. c. An undefined symbolic address or an invalid special character such as close or open parentheses) (is used in the operand.
ER6	A DSA statement has more than ten operands.
ER7	A DSB statement has the second operand missing.
ER8	a. A DC, DSC, DAC, or DNB has a specified length greater than 50. b. A DC, DSC, or DAC statement has no constant specified. c. A DC or DSC has a specified length which is less than the number of digits in the constant itself. d. A DAC statement as a specified length not equal to the number of characters in the constant itself.
ER9	The symbol table is full.
ER10	A duplicate label is defined (defined more than once).
ER11	An assembled address is greater than five digits.
ER12	An invalid special character is used as a head character in a HEAD statement.
ER13	A HEAD statement operand contains more than one character.
ER14	A label contains all numerical characters or an invalid special character. The eight invalid special characters are: blank) + $ * - , (

Format of messages

Error messages take the following general form,

LABEL	adjustment count	error code
XXXXXX	+ XXXX	ERn

where LABEL refers to the last defined label and the "adjustment count" refers to the number of statements between that label and the statement in error. If the first statement of a source program contains the label START, and the second statement has an error "ER1," the following message will be typed:

START + 0001 ER1

If the second statement has the label XYZ, the message still would appear as START + 0001, not as XYZ + 0000.

The messages will appear in the form just shown during pass 1 or pass 2, if program switch 1 is off. If program switch 1 is on during the second pass, only the error code "ERn" will be typed opposite the statement in error, at the right-hand side of the page.

Error Correction

Alternative procedures for few or many errors

As stated earlier, each erroneous statement can be corrected individually after the error message is typed and the machine is stopped, or all statements containing errors can be corrected after the object program is assembled. If there are few errors, the first procedure may be advisable; where there are many errors, it is advisable to correct the source deck at the end of the run and reassemble the program.

Errors detected both passes

Some errors in source statements entered from cards or tape on pass 1 are detected during that pass and, because the same input is used for both passes, will be detected again during pass 2. Therefore, they are corrected twice. Most errors in source statements entered from the typewriter on pass 1 are corrected during that pass. They do not have to be corrected during pass 2, since the output of pass 1 becomes the input to pass 2 and contains the corrected statement.

Program Switch 2 On

Processor stops after an error

With program switch 2 on, the processor stops after typing the error message and

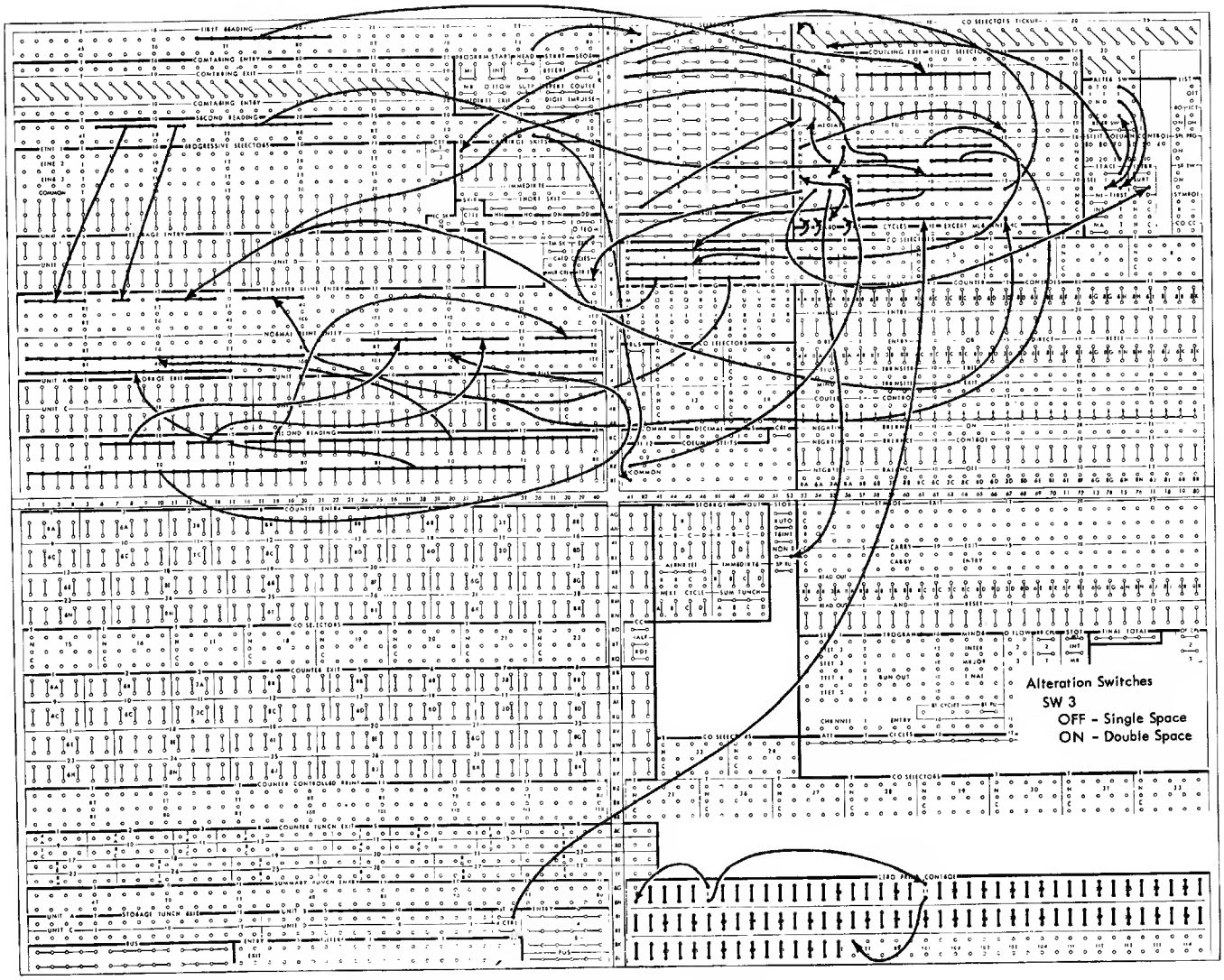


Figure 4B. 407-E8 Control Panel Wiring Diagram for Listing Uncondensed Output

the carriage returns. The operator enters the corrected statement and depresses the release and start keys.

Processor continues after an error

Program Switch 2 Off

With program switch 2 off, the processor does not stop for an error; errors are corrected after assembly. However, errors affect the assembly process as indicated in the following list.

ERROR CODE	ASSEMBLY PROCESS
ER1, ER3	A NOP instruction, 410000000000, is assembled. The label is treated as a blank.
ER2, ER4, ER5, ER11	The operand is assembled as 00000 (zero) address.
ER6	The first ten operands are assembled; those over ten are ignored.
ER7	The statement is assembled in the same manner as a DS statement with a length of 50.
ER8	If the operation code is: DC — it is assembled in the same manner as a DS statement with a length of 50. DSC — it is assembled in the same manner as a DSS statement with a length of 50. DAC — it is assembled in the same manner as a DAS statement with a length of 50. DNB — it is assembled as a DNB with a length of 50.
ER9, ER10, ER14	The label is treated as blank.
ER12	The head character is replaced by a blank character.
ER13	The first non blank character specified in the operand is used as the head character.

Operating Procedures

Typewriter

With program switch 1 on, the typewriter types each statement, starting at the left margin. After the last character is typed, the typewriter carriage returns and then tabulates to the place where typing of the storage address and assembled instruction should begin. Statements are typed in the format in which they are entered; however, a space is inserted before and after the operation field.

To set up the typewriter, the operator must:

1. Set margins to the extreme right and left positions.
2. Set a tab stop in a position that is located a few spaces to the left of the longest statement.

Switches

Both the parity switch and I/O switch should be placed in STOP position; the overflow switch in PROGRAM position. Program switches for controlling the processor should be set as outlined in Table 17.

NOTE: The 1711 ADC unit should be turned off during assembly on a 1710 system.

Loading the Processor

PAPER TAPE PROCESSOR

1. Thread the processor tape.
2. Depress the reset and insert keys.
3. Enter 360000000300 from the typewriter.
4. Depress the release and start keys.

CARD PROCESSOR

1. Depress the reset key (console).
2. Place the processor deck in the read hopper.
3. Depress the load key (reader).
4. If the processor is not followed by two blank cards, the reader start key must be depressed to complete the loading.
   NOTE: After the processor is loaded, 48 will appear in the operation register.

Processing the Source Program

PASS 1

Pass 1

After the processor is loaded, the program halts. Processing begins with the reading of the first statement of the source program.

Paper Tape Input.

1. Thread the input tape (source program).
2. Depress the start key.

Typewriter Input.

1. Type statement and end with a record mark (=).
2. Depress the release and start keys.
3. Repeat steps 1 and 2 until all statements are entered.

Card Input.

1. Place source program card deck in read hopper.
2. Depress reader start key.
3. Depress start key (console).

The message "END OF PASS 1" is typed out at the completion of pass 1 before the machine halts.

PASS 2

Pass 2

The source program in card or paper tape form that is used as the input to pass 1 is also the input to pass 2. Source statements entered at the typewriter during pass 1 are the output of pass 1 and the input to pass 2.

Paper Tape Input.

1. Thread the input tape (source program).
2. Set program switches for pass 2 (see Table 17).
3. Depress the start key.

Card Input.

1. Place the input cards (source program) in the read hopper.
2. Set program switches for pass 2 (see Table 17).
3. Depress the reader start key and punch start key.
4. Depress the start key.

After pass 2 is completed, the message "LOAD SUBROUTINES" is typed out if subroutines are required by the source programs. If the subroutines are not required, the message "END OF PASS 2" is typed and the symbol table is printed.

NOTE: If the operator wishes to branch to the entry point of either pass 1 or pass 2, he must refer to the program listing (included in the program package) to find the absolute addresses. The symbolic name for the entry point of pass 1 is INIT1; for pass 2 it is INIT1 + 12.

Loading the Subroutine.

To load paper tape subroutines:

1. Thread the subroutines.
2. Depress the start key.

To load card subroutines:

1. Place the subroutine card deck in the read hopper.
2. Depress the reader start key.
3. Depress the start key.

An error that occurs while the subroutines are being loaded cannot be corrected by manual intervention because any information inserted in locations 00000-00099 will cause erroneous address modification.

Enter mantissa length

If the subroutine being loaded is a variable length subroutine, the message "ENTER MANTISSA LENGTH" is typed and the machine halts. The operator enters the 2-digit mantissa length from the typewriter. This 2-digit number may range from $\bar{0}2$ to $\bar{4}5$. A mantissa length of $\bar{0}8$ does not have to be entered. The release and start keys are depressed to resume processing of the subroutines. It is the programmer's responsibility to enter the number (length of mantissa) correctly. Program switch 4 may be used to correct an entry made in error (see PROGRAM SWITCHES).

Typeout of symbol table

Only those subroutines used by the source program are punched out as part of the object program. After the subroutines are processed, the message "END OF PASS 2" is typed out followed by the symbol table. The operator may suppress the symbol table typeout by turning program switch 4 on while the message "END OF PASS 2" is being typed. If switch 4 is turned on while the symbol table is being typed, the typeout is terminated and the program halts.

When the symbol table is typed, the labels and their associated addresses are typed, five to a line. The format of each label is as follows:

Format of typeout

associated address	label
XXXXX	*XXXXXX

where the leftmost position of the label contains the head character (assigned either by the processor or the programmer). Any true six-character label is indicated by an asterisk in the position next to the label. In this case, all six characters of the label are true label characters without a head character.

Asterisk indicates six-character label

Assembling further source programs

If at the end of passes 1 and 2, it is desired to assemble other source programs, this can be done without reloading the processor.

Condensed deck after pass 2

After pass 2 is completed by the card processor, the programmer may obtain a condensed object program deck by:

1. Turning console program switch 3 on.
2. Placing the source cards in the read hopper.
3. Depressing the reader start key and punch start key.
4. Depressing the start key.

Pre-editing the Source Program

In some instances where a large source program is to be assembled, the programmer may choose to perform a pre-edit prior to assembling the object program. For a pre-edit, normally only error messages are typed for passes 1 and 2 and no listing is typed or object program is punched. However, the two loader cards and arithmetic tables are punched but they can be discarded. For this reason pre-edit data may be obtained in less time than is required for normal assembly. The error listings from the pre-edit enable the programmer to correct the source progam prior to assembling the object program.

No listing made or object program punched during pre-edit

The operating procedure is the same for passes 1 and 2 as that described earlier for normal assembly of an object program except that program switches 3 and 4 (see Table 17) must be on during the second pass.

Switch settings

Special Procedures for the 1620/1710 Two-Pass Processor

This section describes three special procedures or routines that may be used with the two-pass processor. These routines are not a part of the processor itself but are additional aids for the SPS user. The first routine provides a method of making changes to a condensed object deck without need to reassemble the source program. The second routine modifies a tape or card processor to allow it to use additional storage and the third routine describes the condenser program that is used to condense an uncondensed object deck.

Condensed Object Deck Alterations

Patching

While testing an object program, it is often necessary to change or patch some of the original instructions. To do this, the SPS source deck and the condensed deck must be updated each time an instruction is changed. If the SPS output is an uncondensed deck that is later condensed, it is necessary to update three card decks (the source, uncondensed, and condensed decks). The procedure described here provides an orderly, rapid, and accurate means of correcting the source program.

Instruction in Machine Language	SPS					Actual Machine Location	Actual Machine Location +12	Patch Number
	Page	Line	Label	OP Code	Operands			
Column 1-12	14-15	16-18	19-24	25-28	29-61	65-69	70-74	78-80
One Instruction Per Card Patch Cards Are To Be Placed In Front Of The Last Seven Cards Of The Condensed Deck.								

Figure 5. Patch Card Coding Sheet

Patch Card and Coding Sheet

When an instruction requires correction, the corrected machine language instruction as well as the SPS coding should be recorded on a patch coding sheet. The coding sheet shown in Figure 5 or a similar coding sheet can be used for this purpose. Each entry on the coding sheet is punched into a prepunched patch card. The *prepunched* data is arranged on the patch card as follows:

Format of patch card

Columns 13	± (0, 2, 8 punches)
62-65	0$\bar{0}$1$\bar{}$
70	$\bar{}$
75	blank
76	$\bar{0}$
77	9

Use of the Patch Cards

Inserted in object deck

Patch cards are placed in front of the last seven cards of the condensed object deck but no cards are removed or changed. The condensed object deck is loaded in the same manner as it was before the patch cards were added.

Corrections to the Source Program

Reproduce patch cards to correct the source program

After the object program is tested, the source program deck can be corrected by selecting patch cards with a code 9 in column 77 from the object deck. Card columns 14 through 61 in the selected cards should be reproduced into columns 1 through 48 of blank cards. The reproduced cards should then be inserted in page and line number sequence into the source program deck, and the cards that are being replaced should be discarded.

Patch cards may then be returned to the condensed object deck or the source deck may be reassembled to produce a new object deck.

Modifying the Two-Pass Processor for Additional Storage

An SPS paper tape processor or card processor can be modified to permit a larger symbol table when the system includes the IBM 1623 Core Storage unit.

For the tape processor, a Modifier Program that duplicates the processor tape and allows the user to make the necessary changes is available. For the card processor, the program can be modified by changing a single card of the processor deck.

After the processor has been modified to use a certain amount of storage, it should not be run on a machine that contains less storage; however, it may be run on a machine with greater storage capacity.

Modify the Tape Processor

LOADING THE MODIFIER PROGRAM

1. Thread the Modifier Program.
2. Depress the reset and insert keys.
3. Enter 360000000300 on the typewriter.
4. Depress the release and start keys.

CHANGING THE PROCESSOR TAPE

1. Thread the processor tape.
2. Turn program switch 1 on.
3. Depress the start key.
4. The message "ENTER MEMORY SIZE" is typed out and the machine halts.
5. Enter the storage capacity in thousands from the typewriter; that is, "40" for 40,000 or "60" for 60,000.
6. Depress the release and start keys.
7. The program halts after the last record is copied.

NOTE: If an additional tape is to be copied, repeat steps 1 through 6.

Modifying the Card Processor

1. Manually select the ninth card from the end of the processor deck. This card contains $\bar{0}$20000$\bar{+}$ in card columns 1-7.
2. Prepare a new card, changing columns 1-7 so as to contain $\bar{0}$40000$\bar{+}$ for 40,000 storage positions or $\bar{0}$60000$\bar{+}$ for 60,000 storage positions.
3. Replace the old card in the processor deck with the new card.

Condenser Program

An uncondensed object program deck may be condensed by using a separate, special condenser program. A condensed deck stores programs in fewer cards than an uncondensed deck, thus shortening the time required to load an object program into storage. The *condenser* program occupies core storage locations 15000 through 17225.

Operating Procedure

Several decks may be condensed in succession

The condenser program deck is placed in the card hopper, followed by the uncondensed object deck. The program is loaded by depressing the reset and load keys. When loading is completed, the machine halts; depressing the start key (console) causes the condenser program to begin execution. After the input is processed and a condensed output deck is produced, the machine again halts. Each additional input deck can be condensed by depressing the start key (console), thus causing the program to resume processing.

Condensed Output Deck

Card format

The first two cards of the condensed output deck are loader cards and the last seven are arithmetic table and control cards. In between these are the condensed cards containing instructions or constants. The card format of the condensed object program deck is described in the 1620/1710 Two-Pass Processor section.

7090 Processor for Assembling 1620/1710 Programs

The 7090 processor has been designed to operate on a 709 or 7090 with a minimum of 32,000 storage positions, two channels, and ten tapes. The processor runs under control of the IBSOS Monitor.

Compatible with 1620/1710

With few exceptions, the 7090 processor is completely compatible with the 1620/1710 processor.

Exceptions

1. *Flagged Digits.* Peripheral equipment cannot print flags in the program listing because of inherent machine limitations. Flagged digits appear as letters. Special attention should be paid to the letter "O" which is equivalent to a flagged 6, but could easily be mistaken for a zero.

The following shows the value of the flagged digits and their equivalent symbols as they appear on an output listing:

Flagged Digit	Equivalent Symbol
0 (zero)	— (minus sign)
1	J
2	K
3	L
4	M
5	N
6	O
7	P
8	Q
9	R

2. *Diagnostics.* The 7090 and 1620/1710 processors have different diagnostic procedures. Diagnostic messages are not listed here because of their number. Each message indicates the type of error and the manner in which it is handled by the processor.

Magnetic tape input

Input data cards are of the standard SPS format, as illustrated in Figure 2. Input to the 7090 processor is from tape only, prepared in an off-line card-to-tape operation (IBM 714 or IBM 1401). Each program to be assembled must be preceded by two cards. The first is an identification card containing the following information:

Identification card

Columns 1 — 7, 9 combination punch
2-72 — identification data

This data will be printed out as a header on each page of the printed output listing and may contain such information as the program name, programmer, date, etc. The identification card will also be punched out as part of the assembled output.

Control card

The second card, a control card, contains:

Columns 1 — 7, 9 combination punch
11-12 — mantissa length (02 to 45 digits) for variable length subroutines only

Columns (contd.)	14	subroutine type, 1 = fixed length subroutine without divide feature 2 = fixed length subroutine with divide feature 3 = variable length subroutine without divide or floating point features 4 = variable length with divide feature and without floating point feature 5 = variable length subroutine with divide feature and floating point feature
	15	format, blank = input/output in tape format non-blank = input/output in 1622 card format
	16	core size, 1 = 20,000 positions 2 = 40,000 positions 3 = 60,000 positions

Each program must be followed by one blank card, and the last program must be followed by two or more blank cards. The final card of the deck, which is the control card for the IB SOS monitor, must be a PAUSE or STOP card.

Pass 1

The assembly requires two passes. The first pass generates the symbol table and the equivalent storage addresses. The symbol table accommodates a maximum of 3,000 symbols. All labels are checked for legality and unique definition. Declarative statements are checked to see that their length has been correctly specified. The original statement and computed data are written out on an intermediate tape.

Pass 2

The second pass uses the information compiled by the first pass to complete the assembly. A listing of the original input data and the assembled instructions is written on tape to be listed off-line. All error messages detected during the first and second passes are printed out and precede the statement that is in error. These messages can be readily identified by five asterisks to the left of the statement.

All undefined and doubly-defined symbols are printed out following the program listing. The symbol table is also listed.

Magnetic tape output converted to cards off-line

The assembled program is written on another tape for punching off-line. In the case of a tape machine, these cards, with the exception of the identification card, can be used on the IBM 063 or IBM 870 to produce the desired program tape. If the card machine is specified, the output cards can be entered directly into the 1710.

1000 statements per minute

The 7090 processor will assemble approximately 1,000 SPS statements per minute.

Appendix: Sample Program Prepared by 1620/1710 Processor

This sample program[1] is written to demonstrate certain principles of statement writing and to show the assembly of instructions. It represents a numerical integration program that calculates

$$\int_0^1 \sqrt{3x^2}\ \text{arcsine}\ x\ dx$$

The area under the curve $\sqrt{3x^2}$ arcsine x is desired in the interval $0 \leq x \leq 1$.

$$\text{AREA} = \int_0^1 \sqrt{3x^2}\ \text{arcsine}\ x\ dx$$

The arcsine is computed using Hastings' approximation:[2]

$$\text{arcsine}\ x = \frac{\pi}{2} - \sqrt{1-x} \cdot S(x)$$
$$0 \leq x \leq 1$$

Approximation:

$$S(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5$$

$a_0 = 1.5707, 95207$
$a_1 = -.2145, 12362$
$a_2 = .0878, 76311$
$a_3 = -.0449, 58884$
$a_4 = .0193, 49939$
$a_5 = -.0043, 37769$

To facilitate programming on the 1620, the integral is computed by Simpson's rule:

$$\int_0^1 F(x)dx = \frac{\Delta x}{3}\ [F_0 + F_n + 4(F_1 + F_3 + \ldots + F_{n-1}) + 2(F_2 + F_4 + \ldots + F_{n-2})]$$

where $F(x) = \sqrt{3x^2} \cdot \left[\frac{\pi}{2} - \sqrt{1-x} \cdot S(x)\right]$
n is an even integer
$\Delta x = \frac{1}{n}$
$F_i = F(x_i)$, $i = 0,1,2, \ldots, n$
$x_i = i \cdot \Delta x$

The source program coding sheets and the printed 407 output listing for this sample program are shown in Figures 7 and 8, which appear on subsequent pages.

[1]Contributed by Dr. John H. Duffin, Engineering Department, San Jose State College, 1961.
[2]Hastings, Cecil, Jr. The Rand Corporation. Approximations for Digital Computers. Princeton University Press, Princeton, New Jersey, 1955, p. 161.

IBM

1620/1710 Symbolic Programming System
Coding Sheet

Program: Numerical Integration (Sample Program) Date: Page No. 01 of 8

Routine: Programmer:

Line	Label	Operation	Operands & Remarks
0,1,0	*, THIS	PRO	GRAM COMPUTES THE AREA UNDER THE CURVE SQRT(3-X**2) * ARCSINE
0,2,0	*, WHERE X	LIES BETWEEN 0 AND 1. THE AREA IS COMPUTED BY SIMPSONS RULE
0,3,0	*, FOR	NUMERICAL INTEGRATION. THE AREA IS EVALUATED USING THREE
0,4,0	*, DIFF	ERENT VALUES FOR DELTAX. THEY ARE .01, .005, AND .0025.
0,5,0		DORG	1730
0,6,0	START	TF	DELTAX X 7 TRANSMIT VALUE OF INCREMENT
0,7,0		TF	AREA Z -3
0,8,0		TF	XSUBN UNIT
0,9,0		TDM	SW3+1 1 SET SW3 OFF
1,0,0		TDM	SW2+1 1 SET SW2 OFF
1,1,0		TDM	SW1+1 1 SET SW1 OFF
1,2,0		TR	ASUBN-9 CONST-9 TRANSMIT ASUB5 TO ASUB0
1,3,0		TF	PSIX ASUBN
1,4,0	ASINE	M	PSIX XSUBN
1,5,0		SF	84
1,6,0		BNF	*+2 *L 99
1,7,0		SF	93
1,8,0		TF	PSIX 93
1,9,0		TR	ASUBN-9 ASUBN+1
2,0,0		A	PSIX ASUBN
2,1,0		BNR	ASINE ASUBN+1
2,2,0		BNC,1	CONTA
2,3,0		TD	POLY+48 PSIX-9
2,4,0		TD	POLY+52 PSIX-8
2,5,0		TD	POLY+54 PSIX-7
2,6,0		TD	POLY+56 PSIX-6

IBM

1620/1710 Symbolic Programming System
Coding Sheet

Program: Numerical Integration (Sample Program) Date: Page No. 02 of 8

Routine: Programmer:

Line	Label	Operation	Operands & Remarks
0,1,0		TD	POLY+58 PSIX-5
0,2,0		TD	POLY+60 PSIX-4
0,3,0		TD	POLY+62 PSIX-3
0,4,0		TD	POLY+64 PSIX-2
0,5,0		TD	POLY+66 PSIX-1
0,6,0		TD	POLY+68 PSIX
0,7,0		TD	POLY+12 XSUBN-6
0,8,0		TD	POLY+16 XSUBN-5
0,9,0		TD	POLY+18 XSUBN-4
1,0,0		TD	POLY+20 XSUBN-3
1,1,0		RCTY	
1,2,0		WATY	POLY
1,3,0	CONTA	TF	RADCND UNIT
1,4,0		S	RADCND XSUBN RADICAND # 1-X
1,5,0		TR	RADCND ZNINES -13
1,6,0		BNC,1	CONTB
1,7,0		TD	ARG+42 RADCND-6
1,8,0		TD	ARG+46 RADCND-5
1,9,0		TD	ARG+48 RADCND-4
2,0,0		TD	ARG+50 RADCND-3
2,1,0		TD	ARG+52 RADCND-2
2,2,0		TD	ARG+54 RADCND-1
2,3,0		TD	ARG+56 RADCND
2,4,0		RCTY	
2,5,0		WATY	ARG
2,6,0	CONTB	TF	NINE TWO9

Figure 7. Sample Program Coding Sheets, Part 1

IBM

1620/1710 Symbolic Programming System
Coding Sheet

Program: Numerical Integration (Sample Program) Date: Page No. 03 of 8

Routine: Programmer:

Line	Label	Operation	Operands & Remarks
0,1,0		T,F,	O,D,D,I,N,T,,,O,N,E,O,N,E(E)
0,2,0		B,	*+,2,*,L(E)
0,3,0	R,O,O,T,	A,	O,D,D,I,N,T,-,8,,,T,W,O(E)
0,4,0		S,	R,A,D,C,N,D,+,7,,,O,D,D,I,N,T(E)
0,5,0		B,N,N,	R,O,O,T(E)
0,6,0		A,	R,A,D,C,N,D,+,7,,,O,D,D,I,N,T(E)
0,7,0		T,R,	R,A,D,C,N,D,-,7,,,R,A,D,C,N,D,-,6(E)
0,8,0		S,F,	R,A,D,C,N,D,-,7(E)
0,9,0		S,	O,D,D,I,N,T,-,8,,,N,I,N,E(E)
1,0,0		T,F,	N,I,N,E,,,N,I,N,E,-,1(E)
1,1,0		B,N,F,	R,O,O,T,+,1,*,L,,T,W,O,+,1(E)
1,2,0		T,F,	S,Q,R,T,,,N,I,N,E,S(E)
1,3,0		S,F,	R,A,D,C,N,D,+,1(E)
1,4,0		S,	S,Q,R,T,,,R,A,D,C,N,D,+,6(E)
1,5,0		B,N,C,1	C,O,N,T,C(E)
1,6,0		T,D,	G,E,N,R,T,+,2,4,,,S,Q,R,T,,-,5(E)
1,7,0		T,D,	G,E,N,R,T,+,2,8,,,S,Q,R,T,,-,4(E)
1,8,0		T,D,	G,E,N,R,T,+,3,0,,,S,Q,R,T,,-,3(E)
1,9,0		T,D,	G,E,N,R,T,+,3,2,,,S,Q,R,T,,-,2(E)
2,0,0		T,D,	G,E,N,R,T,+,3,4,,,S,Q,R,T,,-,1(E)
2,1,0		T,D,	G,E,N,R,T,+,3,6,,,S,Q,R,T(E)
2,2,0		R,C,T,Y(E)	
2,3,0		W,A,T,Y	G,E,N,R,T(E)
2,4,0	C,O,N,T,C,	M,	S,Q,R,T,,,P,S,I,X(E)
2,5,0		S,F,	8,5(E)
2,6,0		T,F,	T,E,M,P,1,,,9,4(E)

IBM

1620/1710 Symbolic Programming System
Coding Sheet

Program: Numerical Integration (Sample Program) Date: Page No. 04 of 8

Routine: Programmer:

Line	Label	Operation	Operands & Remarks
0,1,0		B,N,C,1	S,W,1(E)
0,2,0		T,D,	F,U,N,C,T,+,1,0,,,T,E,M,P,1,-,9(E)
0,3,0		T,D,	F,U,N,C,T,+,1,4,,,T,E,M,P,1,-,8(E)
0,4,0		T,D,	F,U,N,C,T,+,1,6,,,T,E,M,P,1,-,7(E)
0,5,0		T,D,	F,U,N,C,T,+,1,8,,,T,E,M,P,1,-,6(E)
0,6,0		T,D,	F,U,N,C,T,+,2,0,,,T,E,M,P,1,-,5(E)
0,7,0		T,D,	F,U,N,C,T,+,2,2,,,T,E,M,P,1,-,4(E)
0,8,0		T,D,	F,U,N,C,T,+,2,4,,,T,E,M,P,1,-,3(E)
0,9,0		T,D,	F,U,N,C,T,+,2,6,,,T,E,M,P,1,-,2(E)
1,0,0		T,D,	F,U,N,C,T,+,2,8,,,T,E,M,P,1,-,1(E)
1,1,0		T,D,	F,U,N,C,T,+,3,0,,,T,E,M,P,1(E)
1,2,0		R,C,T,Y(E)	
1,3,0		W,A,T,Y	F,U,N,C,T(E)
1,4,0	S,W,1,	B,	S,W,2(E)
1,5,0		M,	X,S,U,B,N,,,X,S,U,B,N(E)
1,6,0		S,F,	8,7(E)
1,7,0		T,F,	T,E,M,P,2,,,9,6(E)
1,8,0		M,M,	T,E,M,P,2,,,3,,,1,0(E)
1,9,0		S,F,	9,0(E)
2,0,0		T,F,	R,A,D,C,N,D,,9,6(E)
2,1,0		T,F,	P,S,I,X,,,C,O,N,S,T,+,5,0(E)
2,2,0		S,	P,S,I,X,,,T,E,M,P,1(E)
2,3,0		T,D,M,	S,W,1,+,1,,,9(E)
2,4,0		B,	R,O,O,T,-,1,4,*,L(E)
2,5,0	S,W,2,	B,	O,D,D,V,N(E)
2,6,0		A,	A,R,E,A,,,T,E,M,P,1,,-,4,,,,F,0,+,E,N(E)

Figure 7. Sample Program Coding Sheets, Part 2

IBM

1620/1710 Symbolic Programming System
Coding Sheet

Program: Numerical Integration (Sample Program) Date: Page No. 05 of 8

Routine: Programmer:

Line	Label	Operation	Operands & Remarks
010	*	INITIALIZATION FOR FSUBODD	
020		TF	XSUBN, DELTAX
030		TFM	MULT+11,,4,,10
040		TDM	SW2+1,,9
050		TF	ACCUM, Z
060		TF	TEMP3,, DELTAX
070		A	TEMP3,, TEMP3
080		B	ASINE-3,*L
090	ODDVN	A	ACCUM, TEMP1
100		A	XSUBN, TEMP3
110		C	XSUBN, NINES
120		BNH	ASINE-3,*L
130	MULT	MM	ACCUM
140		SF	8,8
150		A	AREA, 9,5
160	SW3	B	*+6,*L
170	*	INITIALIZATION FOR FSUBEVEN	
180		TFM	MULT+11,,2,,10
190		TF	ACCUM, Z
200		TF	XSUBN, TEMP3
210		TDM	SW3+1,,9
220		B	ASINE-3,*L
230		M	AREA, DELTAX
240		SF	8,8
250		TF	TEMP1,, 9,7
260		M	TEMP1,, THREES

IBM

1620/1710 Symbolic Programming System
Coding Sheet

Program: Numerical Integration (Sample Program) Date: Page No. 06 of 8

Routine: Programmer:

Line	Label	Operation	Operands & Remarks
010		TD	OUTPUT+26,,DELTAX-5
020		TD	OUTPUT+28,,DELTAX-4
030		TD	OUTPUT+30,,DELTAX-3
040		TD	OUTPUT+46,,83
050		TD	OUTPUT+50,,84
060		TD	OUTPUT+52,,85
070		TD	OUTPUT+54,,86
080		TD	OUTPUT+56,,87
090		TD	OUTPUT+58,,88
100		RCTY	
110		WATY	OUTPUT
120		AM	START+11,,7,,10
130		CM	START+11,,X+21
140		BNE	START
150		H	
160	*	AREA DEFINITIONS	
170	DELTAX	DS	7
180	X	DC	7,,100000
190		DC	7,,500000
200		DC	7,,250000
210	AREA	DS	8
220	Z	DC	1,1,,0
230	XSUBN	DS	7
240	UNIT	DC	7,,1000000
250	ASUBN	DSB	10,,6
260		DS	1

Figure 7. Sample Program Coding Sheets, Part 3

IBM

1620/1710 Symbolic Programming System
Coding Sheet

Program: Numerical Integration (Sample Program) Date: Page No. 07 of 8

Routine: Programmer:

Line	Label	Operation	Operands & Remarks
010	CONST	DC	10,-43377699(E)
020		DC	10,193499399(E)
030		DC	10,-44955888484(E)
040		DC	10,8787631,1(E)
050		DC	10,-2145123622(E)
060		DC	11,15707955207@(E)
070	L	DS	,12(E)
080	PSIX	DS	10(E)
090		DS	1(E)
100	RADCND	DS	7(E)
110		DS	13(E)
120	ZNINES	DC	15.999999999@(E)
130	TWO	DS	6(E)
140	NINE	DS	6(E)
150	TWO9	DC	12,200000009000,0(E)
160	ODDINT	DS	14(E)
170	ONEONE	DC	14,10000000000010(E)
180	SQRT	DS	6(E)
190	NINES	DC	6,,999999(E)
200	TEMP1	DS	10(E)
210	TEMP2	DS	10(E)
220	ACCUM	DS	11(E)
230	TEMP3	DS	7(E)
240	THREES	DC	7,33333333(E)
250	POLY	DAC	36,,FOR X#0.000,,,POLYNOMIAL#0.0000000000@(E)
260	ARG	DAC	30,SQUARE ROOT ARGUMENT#0.0000000@(E)

IBM

1620/1710 Symbolic Programming System
Coding Sheet

Program: Numerical Integration (Sample Program) Date: Page No. 08 of 8

Routine: Programmer:

Line	Label	Operation	Operands & Remarks
010	GENRT	DAC	20,,SQUARE ROOT#0.00000@(E)
020	FUNCT	DAC	17,,F(X)#0.0000000000@(E)
030	OUTPUT	DAC	31,,FOR DELTAX#0.000,,,AREA#0.000000@(E)
040		DEND	START(E)
050			
060			
070			
080			
090			
100			
110			
120			
130			
140			
150			
160			
170			
180			
190			
200			

Figure 7. Sample Program Coding Sheets, Part 4

```
                                  * THIS  PRO GRAM COMPUTES THE AREA UNDER THE CURVE SQRT%3X**2¤*ARCSINEX  0002
                                  * WHER E X  LIES BETWEEN 0 AND 1. THE AREA IS COMPUTED BY SIMPSONS RULE  0004
                                  * FOR  NUME RICAL INTEGRATION. THE AREA IS EVALUATED USING THREE         0006
                                  * DIFF EREN T VALUES FOR DELTAX. THEY ARE 0.100. 0.050. AND 0.025.      0008
01732                                    DORG 1732                                                        0010
01732 26 03394 -3401              START  TF   DELTAX.X.7.TRANSMIT VALUE OF INCREMENT                      0012
01744 26 03423 03431                     TF   AREA.Z-3                                                    0014
01756 26 03441 03448                     TF   XSUBN.UNIT                                                  0016
01768 15 03089 00001                     TDM  SW3&1.1..SET SW3 OFF                                        0018
01780 15 02897 00001                     TDM  SW2&1.1..SET SW2 OFF                                        0020
01792 15 02765 00001                     TDM  SW1&1.1..SET SW1 OFF                                        0022
01804 31 03449 03510                     TR   ASUBN-9.CONST-9..TRANSMIT ASUB5 TO ASUB0                    0024
01816 26 03580 03458                     TF   PSIX.ASUBN                                                  0026
01828 23 03580 03441              ASINE  M    PSIX.XSUBN                                                  0028
01840 32 00084 00000                     SF   84                                                          0030
01852 44 01876 00099                     BNF  *&2*L.99                                                    0032
01864 32 00093 00000                     SF   93                                                          0034
01876 26 03580 00093                     TF   PSIX.93                                                     0036
01888 31 03449 03459                     TR   ASUBN-9.ASUBN&1                                             0038
01900 21 03580 03458                     A    PSIX.ASUBN                                                  0040
01912 45 01828 03459                     BNR  ASINE.ASUBN&1                                               0042
01924 47 02128 00100                     BNC1 CONTA                                                       0044
01936 25 03775 03571                     TD   PDLY&4B.PSIX-9                                              0046
01948 25 03779 03572                     TD   PDLY&52.PSIX-8                                              0048
01960 25 03781 03573                     TD   PDLY&54.PSIX-7                                              0050
01972 25 03783 03574                     TD   PDLY&56.PSIX-6                                              0052
01984 25 03785 03575                     TD   PDLY&58.PSIX-5                                              0054
01996 25 03787 03576                     TD   PDLY&60.PSIX-4                                              0056
02008 25 03789 03577                     TD   PDLY&62.PSIX-3                                              0058
02020 25 03791 03578                     TD   PDLY&64.PSIX-2                                              0060
02032 25 03793 03579                     TD   PDLY&66.PSIX-1                                              0062
02044 25 03795 03580                     TD   PDLY&68.PSIX                                                0064
02056 25 03739 03435                     TD   PDLY&12.XSUBN-6                                             0066
02068 25 03743 03436                     TD   PDLY&16.XSUBN-5                                             0068
02080 25 03745 03437                     TD   PDLY&18.XSUBN-4                                             0070
02092 25 03747 03438                     TD   PDLY&20.XSUBN-3                                             0072
02104 34 00000 00102                     RCTY                                                             0074
02116 39 03727 00100                     WATY PDLY                                                        0076
02128 26 03588 03448              CONTA  TF   RADCND.UNIT                                                 0078
02140 22 03588 03441                     S    RADCND.XSUBN..RADICAND # 1-X                                0080
02152 31 03588 03603                     TR   RADCND.ZNINES-13.                                           0082
02164 47 02284 00100                     BNC1 CONTB                                                       0084
02176 25 03841 03582                     TD   ARG&42.RADCND-6                                             0086
02188 25 03845 03583                     TD   ARG&46.RADCND-5                                             0088
02200 25 03847 03584                     TD   ARG&48.RADCND-4                                             0090
02212 25 03849 03585                     TD   ARG&50.RADCND-3                                             0092
02224 25 03851 03586                     TD   ARG&52.RADCND-2                                             0094
02236 25 03853 03587                     TD   ARG&54.RADCND-1                                             0096
02248 25 03855 03588                     TD   ARG&56.RADCND                                               0098
02260 34 00000 00102                     RCTY                                                             0100
02272 39 03799 00100                     WATY ARG                                                         0102
02284 26 03628 03640              CONTB  TF   NINE.TWO9                                                   0104
02296 26 03654 03668                     TF   DODINT.DNEDNE                                              0106
02308 49 02332 00000                     B    *&2*L                                                       0108
02320 21 03646 03622              RDOT   A    ODDINT-8.TWO                                                0110
02332 22 03595 03654                     S    RADCND&7.DODINT                                             0112
02344 46 02320 01300                     BNN  RDOT                                                        0114
02356 21 03595 03654                     A    RADCND&7.DODINT                                             0116
02368 31 03581 03582                     TR   RADCND-7.RADCND-6                                           0118
02380 32 03581 00000                     SF   RADCND-7                                                    0120
02392 22 03646 03628                     S    ODDINT-8.NINE                                               0122
02404 26 03628 03627                     TF   NINE.NINE-1                                                 0124
02416 44 02332 03623                     BNF  RDOT&1*L.TWO&1                                              0126
02428 26 03674 03680                     TF   SQRT.NINES                                                  0128
02440 32 03589 00000                     SF   RADCND&1                                                    0130
02452 22 03674 03594                     S    SQRT.RADCND&6                                               0132
02464 47 02572 00100                     BNC1 CONTC                                                       0134
02476 25 03883 03669                     TD   GENRT&24.SQRT-5                                             0136
02488 25 03887 03670                     TD   GENRT&28.SQRT-4                                             0138
02500 25 03889 03671                     TD   GENRT&30.SQRT-3                                             0140
02512 25 03891 03672                     TD   GENRT&32.SQRT-2                                             0142
02524 25 03893 03673                     TD   GENRT&34.SQRT-1                                             0144
02536 25 03895 03674                     TD   GENRT&36.SQRT                                               0146
02548 34 00000 00102                     RCTY                                                             0148
02560 39 03859 00100                     WATY GENRT                                                       0150
                                                                                    PAGE    1
```

Figure 8. Sample Program Output Listing, Part 1

```
02572 23 03674 03580        CONTC  M     SORT,PS1X                          0152
02584 32 00085 00000               SF    85                                 0154
02596 26 03690 00094               TF    TEMP1,94                           0156
02608 47 02764 00100               BNC1  SW1                                0158
02620 25 03909 03681               TD    FUNCT&10,TEMP1-9                   0160
02632 25 03913 03682               TD    FUNCT&14,TEMP1-8                   0162
02644 25 03915 03683               TD    FUNCT&16,TEMP1-7                   0164
02656 25 03917 03684               TD    FUNCT&18,TEMP1-6                   0166
02668 25 03919 03685               TD    FUNCT&20,TEMP1-5                   0168
02680 25 03921 03686               TD    FUNCT&22,TEMP1-4                   0170
02692 25 03923 03687               TD    FUNCT&24,TEMP1-3                   0172
02704 25 03925 03688               TD    FUNCT&26,TEMP1-2                   0174
02716 25 03927 03689               TD    FUNCT&28,TEMP1-1                   0176
02728 25 03929 03690               TD    FUNCT&30,TEMP1                     0178
02740 34 00000 00102               RCTY                                     0180
02752 39 03899 00100               WATY  FUNCT                              0182
02764 49 02896 00000        SW1    B     SW2                                0184
02776 23 03441 03441               M     XSUBN,XSUBN                        0186
02788 32 00087 00000               SF    87                                 0188
02800 26 03700 00096               TF    TEMP2,96                           0190
02812 13 03700 000-3               MM    TEMP2,3,10                         0192
02824 32 00090 00000               SF    90                                 0194
02836 26 03588 00096               TF    RADCND,96                          0196
02848 26 03580 03569               TF    PS1X,CONST&50                      0198
02860 22 03580 03690               S     PS1X,TEMP1                         0200
02872 15 02765 00009               TDM   SW1&1,9                            0202
02884 49 02152 00000               B     ROOT-14*L                          0204
02896 49 03004 00000        SW2    B     ODDVN                              0206
02908 21 03423 03686               A     AREA,TEMP1-4,,FO&FN                0208
                            * INIT IAL1 ZATION FOR FSUBODD                   0210
02920 26 03441 03394               TF    XSUBN,DELTAX                       0212
02932 16 03063 000-4               TFM   MULT&11,4,10                       0214
02944 15 02897 00009               TDM   SW2&1,9                            0216
02956 26 03711 03434               TF    ACCUM,Z                            0218
02968 26 03718 03394               TF    TEMP3,DELTAX                       0220
02980 21 03718 03718               A     TEMP3,TEMP3                        0222
02992 49 01792 00000               B     AS1NE-3*L                          0224
03004 21 03711 03690        ODDVN  A     ACCUM,TEMP1                        0226
03016 21 03441 03718               A     XSUBN,TEMP3                        0228
03028 24 03441 03680               C     XSUBN,N1NES                        0230
03040 47 01792 01100               BNH   AS1NE-3*L                          0232
03052 13 03711 -0000        MULT   MM    ACCUM                              0234
03064 32 00088 00000               SF    88                                 0236
03076 21 03423 00095               A     AREA,95                            0238
03088 49 03160 00000        SW3    B     *&6*L                              0240
                            * INIT IAL1 ZATION FOR FSUBEVEN                  0242
03100 16 03063 000-2               TFM   MULT&11,2,10                       0244
03112 26 03711 03434               TF    ACCUM,Z                            0246
03124 26 03441 03718               TF    XSUBN,TEMP3                        0248
03136 15 03089 00009               TDM   SW3&1,9                            0250
03148 49 01792 00000               B     AS1NE-3*L                          0252
03160 23 03423 03394               M     AREA,DELTAX                        0254
03172 32 00088 00000               SF    88                                 0256
03184 26 03690 00097               TF    TEMP1,97                           0258
03196 23 03690 03725               M     TEMP1,THREES                       0260
03208 25 03959 03389               TD    OUTPUT&26,DELTAX-5                 0262
03220 25 03961 03390               TD    OUTPUT&28,DELTAX-4                 0264
03232 25 03963 03391               TD    OUTPUT&30,DELTAX-3                 0266
03244 25 03979 00083               TD    OUTPUT&46,83                       0268
03256 25 03983 00084               TD    OUTPUT&50,84                       0270
03268 25 03985 00085               TD    OUTPUT&52,85                       0272
03280 25 03987 00086               TD    OUTPUT&54,86                       0274
03292 25 03989 00087               TD    OUTPUT&56,87                       0276
03304 25 03991 00088               TD    OUTPUT&58,88                       0278
03316 34 00000 00102               RCTY                                     0280
03328 39 03933 00100               WATY  OUTPUT                             0282
03340 11 01743 000-7               AM    START&11,7,10                      0284
03352 14 01743 -3422               CM    START&11,X&21                      0286
03364 47 01732 01200               BNE   START                              0288
03376 48 00000 00000               H                                        0290
                            * AREA  DEF INITIONS                             0292
03394     00007             DELTAX DS    7                                  0294
03401     00007             X      DC    7,100000                           0296
03408     00007                    DC    7,50000                            0299
03415     00007                    DC    7,25000                            0302
03423     00008             AREA   DS    8                                  0305
```

PAGE 2

Figure 8. Sample Program Output Listing, Part 2

```
03434   00011              Z       DC    11.0                                          0307
03441   00007              XSUBN   DS    7                                             0310
03448   00007              UNIT    DC    7.1000000                                     0312
03458   00010 00006        ASUBN   DSB   10.6                                          0315
03509   00001                      DS    1                                             0317
03519   00010              CONST   DC    10.-4337769                                   0319
03529   00010                      DC    10.19349939                                   0322
03539   00010                      DC    10.-44958884                                  0325
03549   00010                      DC    10.87876311                                   0328
03559   00010                      DC    10.-214512362                                 0331
03570   00011                      DC    11.1570795207@                                0334
00012                      L       DS    .12                                           0337
03580   00010              PS1X    DS    10                                            0339
03581   00001                      DS    1                                             0341
03588   00007              RADCND  DS    7                                             0343
03601   00013                      DS    13                                            0345
03616   00015              ZNINES  DC    15.9999999@                                   0347
03622   00006              TWO     DS    6                                             0350
03628   00006              NINE    DS    6                                             0352
03640   00012              TWD9    DC    12.200000090000                               0354
03654   00014              DDDINT  DS    14                                            0357
03668   00014              DNEDNE  DC    14.10000D0D000001                             0359
03674   00006              SQRT    DS    6                                             0362
03680   00006              NINES   DC    6.999999                                      0364
03690   00010              TEMP1   DS    10                                            0367
03700   00010              TEMP2   DS    10                                            0369
03711   00011              ACCUM   DS    11                                            0371
03718   00007              TEMP3   DS    7                                             0373
03725   00007              THREES  DC    7.3333333                                     0375
03727   00036              POLY    DAC   36.FOR X#0.000. POLYNOMIAL#0.000000000@      0378
03799   00030              ARG     DAC   30.SQUARE RDDT ARGUMENT#0.000000@            0382
03859   00020              GENRT   DAC   20.SQUARE RDDT#0.00000@                      0386
03899   00017              FUNCT   DAC   17.F%X□#0.0000000000@                        0389
03933   00031              OUTPUT  DAC   31.FOR DELTAX#0.000. AREA#0.00000@          0392
01732                              DEND  START                                       0396
                                                                        PAGE    3
```

Figure 8. Sample Program Output Listing, Part 3

Index

IBM 1620/1710 Printed in U.S.A. C26-5600-1

IBM

International Business Machines Corporation
Data Processing Division
112 East Post Road, White Plains, New York